Key Value Store

Smău George Robert

Facultatea de Informatică, Universitatea Alexandru Ioan Cuza Iași

1 Introducere

În acest raport tehnic voi prezenta ideea mea de implementare a proiectului Key Value Store. Aplicația trebuie să ofere un mecanism de caching a datelor sub forma de perechi cheie - valoare, să poată stoca datele in memorie la nevoie, să sincronizeze datele cu alți clienți și să ofere un mecanism de "undo" (ex. commit/rollback).

2 Tehnologii Aplicate

Pentru implementare am decis să utilizez o arhitectură de tip server-client. Clienții dețin o copie a datelor și vor face operațiile local iar apoi le vor propaga celorlalți clienți prin server, dacă este cazul.

2.1 Concurența

Serverul folosește TCP pentru a asigura transmiterea completă și ordonată a datelor. Concurența este implementată prin multiplexare pentru un consum minim de memorie și distribuirea ușoară a mesajelor.

2.2 STL

Pentru a ușura implementarea aplicației am ales să folosesc limbajul C++ și resursele oferite de STL. Folosesc o stivă pentru implementarea sistemului de "undo", o coadă cu prioritate pentru ștergerea datelor după expirarea acestora și un map pentru a reține în memorie perechile cheie - valoare.

2.3 Thread

Pe lângă thread-ul principal pe care rulează aplicația, vom folosi un al doilea thread care va avea rol de garbage collector. Deoarece acest lucru introduce riscul de a experimenta fenomenul de data race, alegem să folosim mutex-uri pe toate operațiile asupra map-ului.

2.4 JSON

Pentru a stoca datele persistent vom folosi formatul JSON deoarece este ușor de parsat. Parsarea o facem utilizând librăria https://json.nlohmann.me/ pentru a ușura implementarea.

3 Structura Aplicației

Aplicația este compusă din 3 componente: serverul ce se ocupă de distribuție, clientul care este executabilul cu care utilizatorul interacționează și clasa KeyValueStore pe care am construit-o ca pe un API pentru a putea fi refolosită în alte aplicații.

3.1 Server

Primul client care se conectează la un port va lansa automat în execuție un server pe acel port. Acesta va acționa ca un proces copil ce se va detașa de proecesul părinte și va rula independent atâta timp cât există clienți conectați la el. Acest server are rol de distributor, distribuind orice mesaj primește de la un client la toți ceilalți, fără a procesa datele. Este treaba clientului de a valida datele pe care le primește de/transmite la server.

3.2 Client

Clientul este aplicația cu care interacționează utilizatorul. Acesta se conectează la serverul distributor și parsează datele de la utilizator pentru a le trimite drept comenzi la Key Value Store. Practic clientul urmează următoarea structură

```
// conectarea la server
while(running) {
    read(0, buffer, sizeof(buffer));

    CMDStructure cmd = InputParser(buffer);

    Response resp = KVStore.Handler(cmd, true);

    cout << resp.value << '\n';
}
```

Structura CMDStructure conține informații relevante clasei Key Value Store pentru a executa o comandă data, iar structura Response conține răspunsul și succesul comenzii introduse.

```
struct CMDStructure {
    CMD CMDEnum;
    string key = "";
    string value = "";
    time_t TTL = 0;
};

struct Response {
    string value;
    bool success;
};
```

3.3 Key Value Store

Următoarea structură a clasei Key Value Store a fost gândită astfel încât să poată fi adaptată cât mai ușor la alte aplicații. Comenzile acceptate sunt:

1. **set <key> <value> <TTL>** - introduce în baza de date valoarea *value* corespondentă cheii *key* pentru TTL secunde
2. **get <key>** - returnează valoarea corespondentă cheii *key* sau o eroare altfel
3. **delete <key>** - șterge cheia *key* și valoarea corespondentă acesteia, dacă există
4. **commit** - salvează starea actuală a bazei de date
5. **rollback** - întoarce baza de date la ultima stare salvată, dacă există
6. **deletesaves** - șterge salvarile bazei de date
7. **size** - returnează dimensiunea bazei de date
8. **printall** - afișează toate perechile cheie-valoare din baza de date

Utilizatorul clasei are la dispoziție, pe lângă constructor și destructor, următoarele metode publice:

```
1 friend CMDStructure InputParser(string raw);
2 Response Handler(CMDStructure cmd, bool propagate = false);
```

Prima metodă este o funcție de parsare oferită de API pentru a obține dintr-un string un CMDStructure. A doua primește o structură CMDStructure și calculează rezultatul executând comanda din structura dată. Argumentul opțional *propagate* indică nevoia de a partaja schimbarea cu o altă instanță a clasei sau nu.

Observații

Utilizatorul nu este obligat să folosească funcția InputParser pentru a obține structura CMDStructure. Așadar datele pot fi impachetate inițial în orice format atâta timp cât sunt traduse corect într-o instanță a structurii CMDStructure.

De asemenea se observă că dacă dorim să partajăm datele prin opțiunea *propagate* atunci trebuie să oferim un file descriptor în constructor:

```
1 KeyValueStore(int fd, size_t limit, ostream* stream);
```

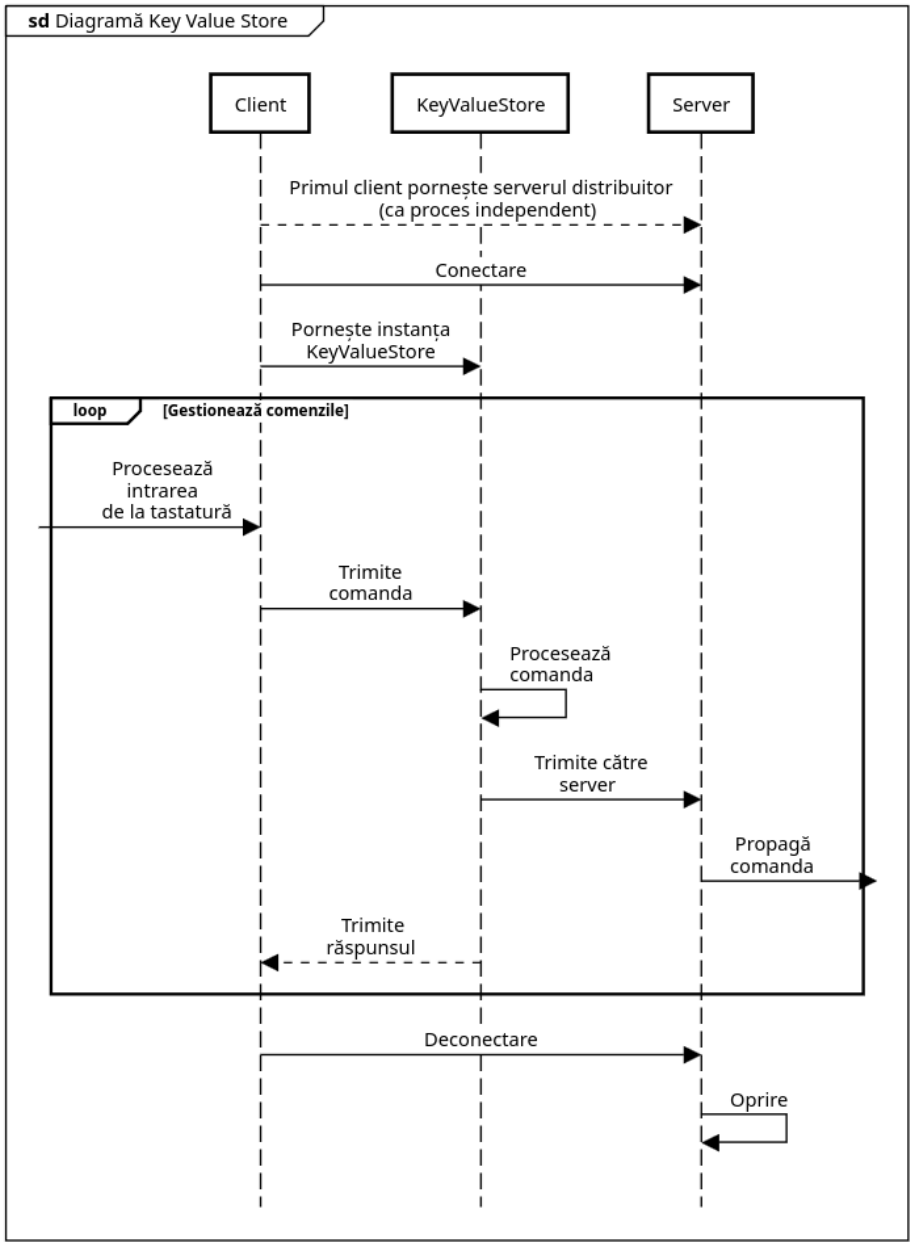

Fig. 1. Structura aplicației.

4 Aspecte de Implementare

4.1 Lansarea în execuție a serverului

Fiecare client primește un port ca parametru și încearcă să se conecteze la acesta.

```
if(bind(socketfd, (const struct sockaddr *)&serverAddr,
    sizeof(serverAddr)) == 0) {
    distributionHandler(socketfd);
} else assert(errno == EADDRINUSE);
```

Dacă portul este ocupat se apelează funcția distributionHandler care lansează un proces copil unde rulează serverul. Procesul este detașat de procesul părinte și rulează atâta timp cât sunt clienți conectați.

```
pid_t pid = fork();
assert(pid != -1);
assert(setsid() != -1);
while(running) {
        assert(select(maxfd+1, &readfds,NULL,NULL,&tv) >= 0);

        if(FD_ISSET(socketfd, &readfds)) {
            int clientfd = accept(socketfd, (struct sockaddr
                *)&clientAddr, (socklen_t *)&clientAddrLen);
            FD_SET(clientfd, &actfds);
            clientCount ++;
        }

        for(int fd = 3; fd <= maxfd; fd ++)
            if(fd != socketfd && FD_ISSET(fd, &readfds)) {
                int bytes = read(fd, buffer, sizeof(buffer));

                if(bytes == 0) {
                    close(fd);
                    FD_CLR(fd, &actfds);
                    clientCount --;
                    if(clientCount == 0) running = false;
                    continue;
                }

                if(bytes < 0) continue;

                for(writefd = 3;writefd <= maxfd;writefd ++)
                    if(writefd != socketfd && writefd != fd)
                    {
                        write(writefd, buffer, bytes);
                    }
            }
    }
```

Apoi, dacă nu există alte erori la bind decât portul ocupat, se continuă execuția clientului.

4.2 Sistemul de debugging / logging

Pentru a putea urmări funcționarea corectă a aplicației am introdus două sisteme: un sistem de debugging în client prin argumentul opțional $-d$ și un sistem de logging pentru server și clasa KeyValueStore. Acesta din urmă creează un fișier în folderul *./logs/* corespondent fiecărei rulări a programului și adaugă informații relevante derulării corecte a aplicației.

```
#define DEBUGMSG(format, ...) if(DEBUG) fprintf(stderr,
    format, ##__VA_ARGS__)
#define LOGMSG(format, ...) fprintf(LOG, format, ##
    __VA_ARGS__)

LOGMSG("[ constructor ] Initialized KVStore\n");
DEBUGMSG("[ client - status ] Connected to server\n");
```

4.3 Declararea comenzilor

Pentru a ușura implementarea de comenzi noi am ales să folosesc următorul macro pentru a îmi genera automat un *enum* și un *vector* $<$ *string* $>$ corespondente comenzilor existente. Totuși funcțiile trebuie declarate manual datorită semnăturilor diferite.

```
#define FOREACH_CMD(FUNC) \
    FUNC(ERROR) \
    FUNC(COMMIT) \
    FUNC(ROLLBACK) \
    FUNC(DELETESAVES) \
    FUNC(SIZE) \
    FUNC(PRINTALL) \
    FUNC(GET) \
    FUNC(DELETE) \
    FUNC(SET) \

#define ENUM(CMD) CMD,
#define NAME(CMD) #CMD,

enum CMD { FOREACH_CMD(ENUM) };

vector<string> CMDEnumToString = { FOREACH_CMD(NAME) };
map<string, CMD> CMDStringToEnum = [] {
    map<string, CMD> m;
    for(int i = 0; i < CMDEnumToString.size(); i ++)
        m[CMDEnumToString[i]] = CMD(i);
    return m;
}();
```

Structurile de date de mai sus ne permit trecerea cu ușurință din forma textuală a unei comenzi în *enum*-ul corespunzător și invers.

4.4 Recycle Bin

O necesitate a acestui proiect este eliminarea automată a cheilor după un anumit timp pe care îl setăm la introducerea lor. Această acțiune este implementată prin funcția $RecycleBin()$ care folosește o coadă cu prioritate pentru a sorta cheile după momentul când trebuie eliminate. Mai exact, în funcția $Set(stringkey, stringvalue, time_tTTL)$ calculăm secunda la care ar trebui să fie eliminată cheia dată și introducem în coada cu prioritate $priority_queue < Entry > recycleBin$ o nouă intrare.

```
struct Entry {
    string key;
    time_t deleteTime;

    // priority queue = max-heap | we reverse the the
        operation to make it a min-heap
    bool operator <(const Entry &a) const {
        return deleteTime > a.deleteTime;
    }
};
```

Cum a fost precizat anterior, funcția $RecycleBin()$ rulează pe un alt thread și este "pornită" în constructor și "oprită" în destructor.

```
// constructor
recyclerThread = thread(&KeyValueStore::RecycleBin, this);

// destructor
recycle = false;
if(recyclerThread.joinable())
    recyclerThread.join();
```

4.5 Funcția $SYNC$

Un posibil scenariu de utilizare a aplicației noastre este:

Se conectează clientul 1.
Clientul 1 execută niște comenzi care modifică starea curentă a bazei de date.
Se conecteaza clientul 2.

În acest scenariu clientul 2 nu are acces la modificările făcute înainte conectării, ci doar la o bază de date. Acesta este motivul pentru care am introdus funcția $SYNC$ ce facilitează sincronizarea unui client nou cu cei deja conectați.

Aceasta este implementată prin solicitarea serverului distributor să gasească un partener de sincronizare. Apoi, clientul care trimite datele va interpreta toata baza de date drept comenzi SET și PUSH pe care le va trimite clientului care dorește să se sincronizeze.

De menționat că această funcționalitate este costisitoare datorită lucrului cu stive.

4.6 Mecanismul *PUSH/POP*

Pentru a oferi utilizatorului posibilitatea de a salva anumite stări ale bazei de date am introdus funcțiile *PUSH* și *POP*.

Funcția *PUSH* salvează starea curentă a bazei de date.

```
Response Push() {
    LOGMSG("[ push ] Adding a new recyler bin\n");
    recycleBin.push(recycleBin.top());

    LOGMSG("[ push ] Saving cache\n");
    size.push(size.top());
    cacheSaves.push(cacheSaves.top());

    LOGMSG("[ push ] Creating new json file\n");
    string storagepath = "./temp/" + to_string(cacheSaves.
        size()) + "-" + string(timeString) + ".json";
    ofstream storage(storagepath);

    string laststoragepath = "./temp/" + to_string(cacheSaves
        .size() - 1) + "-" + string(timeString) + ".json";
    ifstream laststorage(laststoragepath);

    json object = json::object();
    laststorage >> object;

    storage << object.dump(4);

    return { "Cache state saved", true };
}
```

Funcția *POP* șterge starea actuală, revenind la ultima salvare.

```
Response Pop() {
    if(cacheSaves.size() < 2) {
        return { "No saved state to reverse to", false };
    }

    recycleBin.pop();

    string storagepath = "./temp/" + to_string(cacheSaves.
        size()) + "-" + string(timeString) + ".json";
    remove(storagepath.c_str());

    cacheSaves.pop();
    size.pop();

    return { "Cache reversed to last saved state", true };
}
```

Acest mecanism a necesitat implementarea unor stive pentru datele salvate în memorie, sistemul de reciclare dar și creearea mai multor fișiere pentru stocarea persistentă.

```
1 stack<priority_queue<Entry>> recycleBin;
2 stack<map<string,string>> cacheSaves;
```

Mai mult, am adăugat o funcție $DELETESAVES$ care șterge salvările pentru a putea curăța memoria folosită.

5 Concluzii

Proiectul este cât se poate de complex, acesta necesitând îmbinarea unor mecanisme diverse în timp real și distribuit: sistemul de salvare, baza de date propriu zisă, sincronizarea între utilizatori, mecanismul de ștergere automată a datelor. Obiectivele proiectului au fost îndeplinite, aplicația suportând introducerea de perechi cheie-valoare cu uun timp de viață, obținerea valorii ce corespunde unei chei, ștergerea unei perechi, salvarea persistentă a datelor la nevoie și nu numai.

Mai mult, am introdus funcționalitatea de sincronizare ce permite unui utilizator nou să sincronizeze datele cu cei deja conectați la server înainte de își începe treaba, obținând întreaga stare a bazei de date, cu tot cu timpii de ștergere și stiva de salvări.

5.1 Posibile Îmbunătățiri

Deși aplicația este funcțională, există aspecte ale acesteia care pot fi îmbunătățite.

Interfața Momentan aplicația poate fi accesată prin intermediul unui terminal. Deși acest lucru ușurează introducerea de comenzi, un minus este faptul că utilizatorului îi este greu de urmărit schimbările asupra bazei de date. O îmbunătățire ar fi adăugarea unei interfețe grafice pentru a ușura acest aspect.

Politica de înlocuire a memoriei Un aspect important al unei baze de date este viteza cu care poate lucra cu datele. Așadar, o optimizare a aplicației ar putea fi introducerea unui mecanism de înlocuire a memoriei care să fie responsabil de interschimbarea unor perechi de date salvate în memorie cu alte perechi de date salvate în fișiere. Aceasta ar putea crește viteza de acces la date, ceea ce ar rezulta într-o experiență mai bună a utilizatorului.

Arhitectura Deși aplicația este funcțională, arhitectura aleasă nu permite cu ușurință dezvoltarea ei ulterioră. Modificarea arhitecturii într-o formă mai modulară ar ajuta la menținerea și îmbunătățirea proiectului pe termen lung.

References

1. Linux Man Pages, https://linux.die.net/man/
2. C++ Documentation, https://cplusplus.com/doc/
3. Computer Networks Course - Faculty of Computer Science, https://edu.info.uaic.ro/computer-networks/index.php
4. Documentation Latex Format, (https://www.springer.com/gp/computer-science/lncs/conference-proceedings-guidelines
5. Aplication Diagram Software, https://sequencediagram.org
6. JSON for Modern C++, https://json.nlohmann.me/