Tema 2 - Load Balancer

Responsabili:

- Andrei Topală [mailto:topala.andrei@gmail.com]
- Armand Nicolicioiu [mailto:armand.nicolicioiu@gmail.com]
- Iulia Corici [mailto:coriciiulia76@gmail.com]

- Data publicării: **14.04.2021**
- Deadline **HARD**: **09.05.2021 23:55:00**

Actualizări

17 aprilie 2021 22:52: Corectare test12-16.ref

22 aprilie 2021 10:00: Corectare checker

22 aprilie 2021 20:55: Corectare schelet (main.c) + test5,12-16.ref

22 aprilie 2021 23:30: Adăugare instanţă de upload pe vmchecker.

Obiective

În urma realizării acestei teme veţi:

- Învăţa aplicaţii utile şi populare ale hash-urilor existente în literatură.
- Exersa implementarea unui sistem mai complex urmând o descriere detaliată a fiecărei componente.
- Căpăta intuiţie despre limitările unui volum mare de date şi despre sistemele distribuite.

Introducere

Roby, mare antreprenor, doreşte să lanseze o companie de e-Commerce care să rivalizeze Amazon. O primă problemă pe care o întâmpină este aceea de a gestiona toate produsele care vor fi puse la dispoziţie pe site.

Deoarece compania lui Roby deţine un număr foarte mare de produse, acestea nu vor putea fi stocate pe un singur server. De aceea, el va folosi un sistem distribuit în care dorește împărțirea uniformă de produse pe fiecare server. Într-un sistem dinamic precum acesta trebuie ținut cont de faptul că pe parcurs pot fi adăugate servere noi sau oprite servere vechi. Un server nou trebuie să preia o parte din load-ul existent în sistem, iar un server scos va trebui să își transfere produsele către alte servere rămase disponibile.

Cerinta

Scopul vostru este de a implementa un **Load Balancer** folosind **Consistent Hashing.** Acesta este un mecanism frecvent utilizat în cadrul sistemelor distribuite şi are avantajul de a îndeplini *minimal disruption constraint*, adică minimizarea numărului de transferuri necesare atunci când un server este oprit sau unul este pornit. Mai exact, când un server este oprit, doar obiectele aflate pe acesta trebuie redistribuite către servere apropiate. Analog, când un nou server este adugat, va prelua obiecte doar de la un număr limitat de servere, cele vecine.

Load Balancer este componenta care are rolul de a dirija **uniform** traficul către un set de servere cu o putere limitata de procesare. Acesta are misiunea de a asigura că toate serverele stochează și procesează un volum similar date pentru a maximiza eficienta întregului sistem. În cazul nostru, **obiecte din sistem vor fi perechi `<cheie, valoare>`** unde cheia va fi id-ul unui produs iar valoarea va salva detaliile despre produs (pentru simplitate doar un string cu numele, dar poate fi extins la o structura ce conține nume, preț, specificații, rating, etc). Load balancer-ul e responsabil de a decide pe ce server va fi salvat un obiect în funcție de cheia acestuia. Mecanismul eficient prin care acesta mapează un obiect <cheie, valoare> unui server_id poartă numele de **Consistent Hashing** şi este descris în detaliu într-o secțiune următoare.

O metoda simpla de a implementa un load balancer se poate realiza prin asignarea unei sarcini către un server responsabil folosind următoarea formula: `server_id = hash(data) % NUM_SERVERS`

Astfel, load balancer-ul îşi îndeplineşte scopul şi reuşeşte să distribuie uniform datele spre toate cele NUM_SERVERS servere din sistem. În schimb, dezavantajul acestei metode este faptul că într-un sistem real numărul de servere nu este niciodata constant. Dacă unul din servere dispare, din cauza acestei metode simpliste ar trebui să redirijam toate datele din sistem.

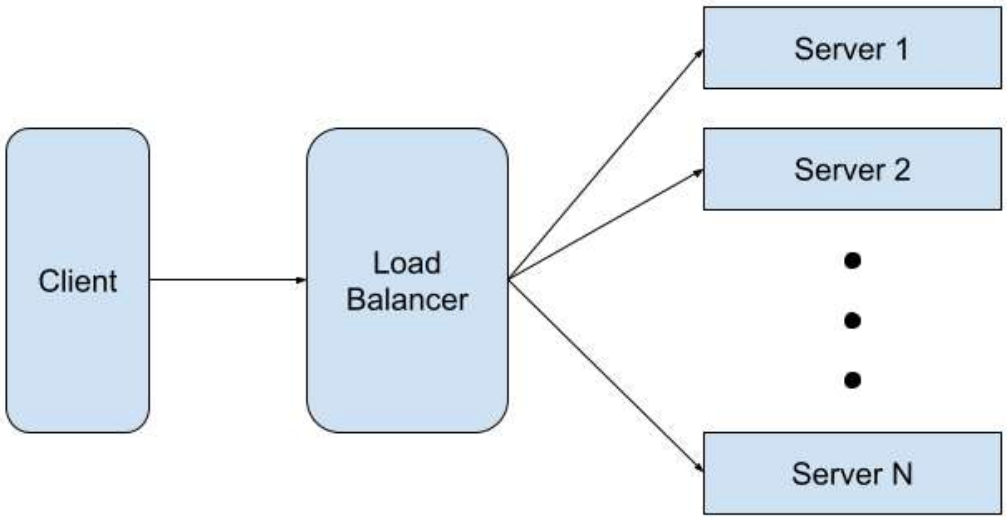
Exemplu:

1. `NUM_SERVERS = 10`
2. Vream sa stocam cheia "123" cu valoarea "iPhone 12" in sistem. hash("123") = 0x65b71f5b. Serverul care va stoca această informaţie va fi: `server_id = 0x65b71f5b % 10 = 1`
3. Un server dispare. `NUM_SERVERS = 9` Vor trebui verificate toate obiectele din sistem pentru că hash(data) % NUM_SERVERS va avea altă valoare față de cea precedentă.
4. Pentru cheia "123" adăugată anterior avem: `server_id = hash("123") % NUM_SERVERS = 0x65b71f5b % 9 = 4` ⇒ În trecut era salvată pe serverul 1, dar acum va trebui mutată pe serverul cu id-ul 4.
5. De asemenea, și id-urile serverelor se vor shifta pentru a avea indici consecutivi.

Observăm că această metodă este foarte ineficientă.

Detaliile Sistemului

În cadrul acestei teme, vă propunem organizarea întregului task sub forma unei ierarhii cu următoarele componente: un client, un load balancer şi mai multe servere.



Client - Are la dispozitie 2 operaţii:

1. client_store(char* key, char* value): Stochează un produs (value) în sistem şi îl asociază unui ID (key).
2. client_retrieve(char* key): Returnează numele produsului asociat unui ID (key).

Load Balancer (Main Server) - Are la dispoziţie 4 operaţii:

1. loader_store(char* key, char* value): Stochează un produs (cheia - ID, valoarea - numele produsului) pe unul dintre serverele disponibile folosind Consistent Hashing.
2. loader_retrieve(char* key): Calculează pe ce server este stocat key şi îi extrage valoarea.
3. loader_add_server(int server_id): Adaugă un nou server în sistem şi rebalansează obiectele.
4. loader_remove_server(int server_id): Scoate un server din sistem şi rebalansează obiectele.

Server - Are la dispoziţei 3 operaţii:

1. server_store(char* key, char* value): Stochează într-un Hashtable datele primite de la Load Balancer.
2. server_retrieve(char* key): Returnează valoarea asociată lui key din Hashtable.
3. server_remove(char* key): Şterge o intrare din Hashtable.

Puteţi adăuga şi alte funcţii noi cât timp nu schimbaţi semnăturile celor deja existente în schelet. De exemplu, dacă este necesar puteţi adăuga o funcţie server_retrieve_all care să returneze toate cheile & valorile de pe un server.

În fișierul `main.c` din schelet este deja implementată partea ce ține de client si modul în care acesta apelează sistemul distribuit, dar și partea de gestiune a sistemului ce va apela funcțiile de pornire si oprire a serverelor atunci când este cazul. Trebuie să implementați doar funcționalitățile din `load_balancer.c` si din `server.c`, entry point-ul către sistemul distribuit fiind prin **Load Balancer** (conform diagramei de mai sus), iar toate apelurile către acesta fiind deja implementate în schelet.

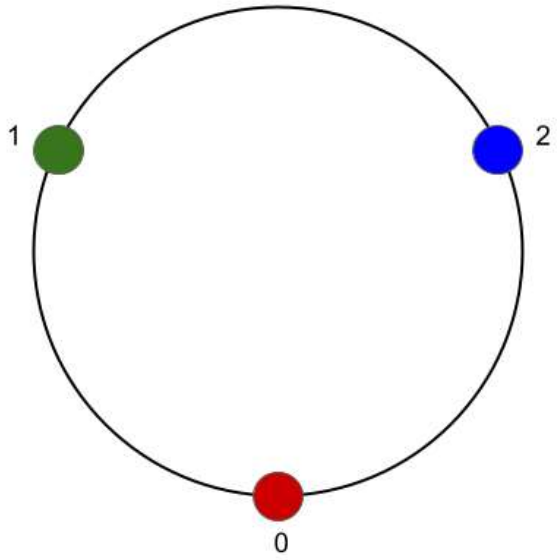
Consistent Hashing

Consistent Hashing este o metoda de hashing distribuit prin care atunci când tabelul este redimensionat, se vor remapa în medie doar `n / m` chei, unde *n* este numărul curent de chei, iar *m* este numărul de slot-uri (în cazul nostru servere). În implementare, se va folosi un cerc imaginar numit *hash ring* pe care sunt mapate atât cheile obiectelor, cât și id-urile serverlor. Această mapare se realizează cu ajutorul unei funcții de hashing cu valori între MIN_HASH și MAX_HASH, punctele din acest interval fiind distribuite la distante egale în intervalul logic [0°, 360°] de pe cercul imaginar.

Fiecare obiect trebuie sa aparţină unui singur server. Astfel, **serverul responsabil să stocheze un anumit obiect va fi cel mai apropiat de pe hash ring în direcţia acelor de ceas**.

Să presupunem că avem următorul set de servere ce urmează să primească obiecte şi rezultatul funcției de hashing după ce a fost aplicată pe id-ul acestora:

ID Server	Hash
2	2269549488
0	5572014558
1	8077113362

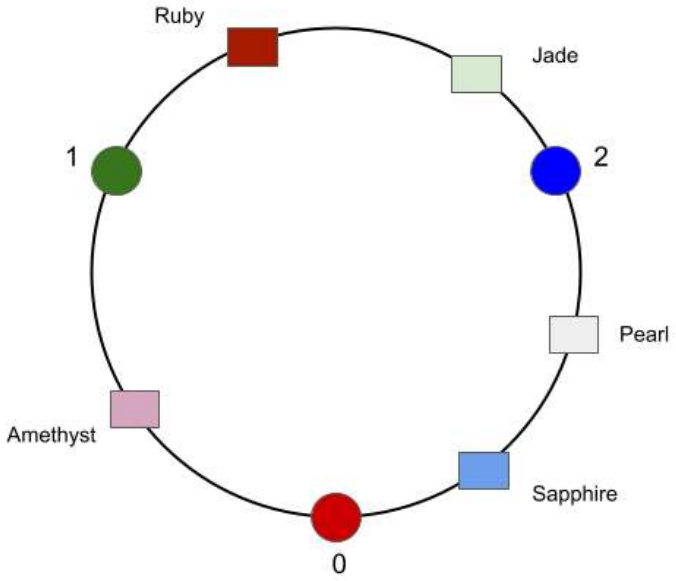


Acum vom adăuga în sistem un set de obiecte ce urmează a fi distribuite către serverele deja existente. În momentul în care un obiect este adăugat în sistem, va trebui să căutăm primul server a cărui funcţie hash este mai "mare" (în sensul acelor de ceas, atenţie la logica circulară) decât funcţia hash a obiectului.

Pentru a produce interogări cât mai eficiente, hash ring-ul va fi reţinut în memorie ca un array ordonat crescător de etichete de servere. Ordonarea se va face în funcţie de hash, iar în cazul în care 2 etichete de servere au aceeaşi valoare hash, se va sorta crescător după ID-ul serverului.

Atentie! În implementare obiectele **NU** vor fi puse pe hash ring. Ele vor fi stocate pe servere. Reprezentările grafice vor pune obiectele pe hash ring pentru a explica mai uşor conceptul şi a putea vizualiza care este cel mai apropiat server.

Key / ID Server	Hash	Stored on
Jade	1633428562	2
2	2269549488	-
Pearl	3421657995	0
Sapphire	5000799124	0
0	5572014558	-
Amethyst	7594634739	1
1	8077113362	-
Ruby	9787173343	2



Replici

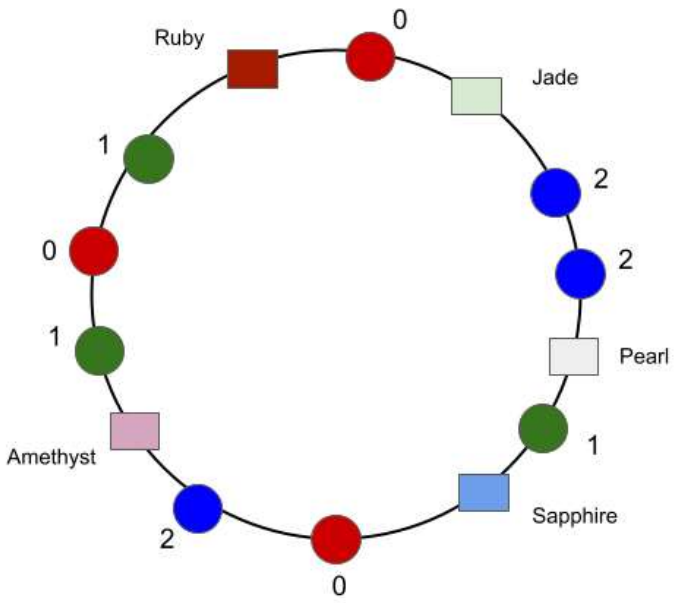
În practică, pentru a ne asigura că obectele sunt distribuite cât mai uniform pe servere se foloseşte următorul artificiu: Fiecare server va fi adaugat de mai multe ori pe hash ring (Aceste servere se vor numi "replici"). Acest mecanism se va realiza prin asocierea unei etichete artificiale fiecarei replici, plecând de la id-ul server-ului de bază:

```
eticheta = replica_id * 10^5  + server_id;
```

De exemplu, replica 2 a serverului cu id-ul 24 ⇒ eticheta = 200024
Vom lucra cu cel mult 99999 servere, deci etichetele vor fi unic determinate.

În implementarea noastră, fiecare server va fi replicat de fix 3 ori (în total va fi reprezentat în 3 puncte).

Key / Tag Server	Hash	Stored on
100000	1093130520	-
Jade	1633428562	2
000002	2269549488	-
200002	2717580620	-
Pearl	3421657995	1
200001	4633428562	-
Sapphire	5000799124	0
000000	5572014558	-
100002	6252163898	-
Amethyst	7594634739	1
100001	7613173320	-
200000	7893130520	-
000001	8077113362	-
Ruby	9787173343	0

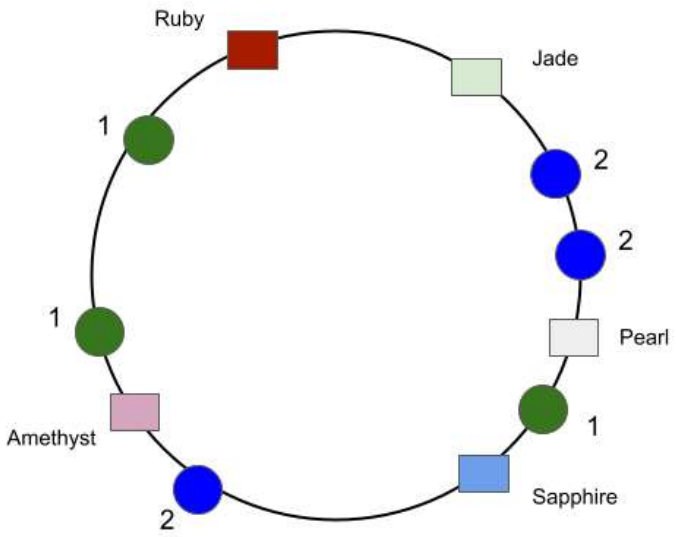


Eliminare

În cazul în care unul din servere este eliminat din sistem, toate replicile sale sunt eliminate de pe hash ring, iar obiectele salvate pe acestea sunt remapate la cele mai apropiate servere ramase (în sensul acelor de ceas).

În exemplul nostru vom presupune că serverul cu id 0 va fi eliminat. În acest moment obiectele "Sapphire" şi "Ruby" vor fi redistribuite

Key / Tag Server	Hash	Stored on
Jade	1633428562	2
000002	2269549488	-
200002	2717580620	-
Pearl	3421657995	1
200001	4633428562	-
Sapphire	5000799124	2
100002	6252163898	-
Amethyst	7594634739	1
100001	7613173320	-
000001	8077113362	-
Ruby	9787173343	2

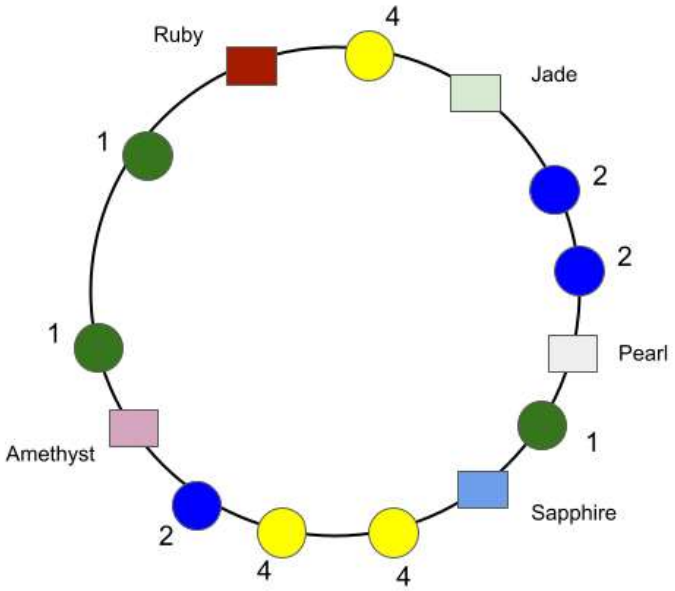


Adăugare

În cazul adăugării unui nou server în sistem, se vor lua toate obiectele de pe serverele vecine (succesoare în sensul acelor de ceas) si se va verifica dacă vor fi remapate către serverul nou sau nu. Dacă un obiect trebuie să fie mapat pe serverul nou, valoarea sa va fi transferată de pe serverul vechi, iar serverul vechi o va şterge.

Să presupunem că se adaugă un server cu id 4. Serverele care vor fi vecine celui cu id 4 (oricare replică a lui) vor trebui să îşi redistribuie obiectele:

Key / Tag Server	Hash	Stored on
Jade	1633428562	2
000002	2269549488	-
200002	2717580620	-
Pearl	3421657995	1
200001	4633428562	-
Sapphire	5000799124	4
000004	5421603348	-
200004	6017580621	-
100002	6252163898	-
Amethyst	7594634739	1
100001	7613173320	-
000001	8077113362	-
Ruby	9787173343	4
100004	9945918562	-



În situaţia de mai sus, serverul 2 a trebuit să distribuie serverului 4 obiectele "Ruby" si "Sapphire". Este o coincidenţă că ambele obiecte redistribuite vin de la acelasi server, puteau veni de la replici ale unor servere diferite.

Implementare

În scheletul de cod veţi avea de implementat funcţiile din fişierele `load_balancer.c` şi `server.c`. Citiţi cu atenţie semnăturile acestor funcţii! Scheletul vine deja cu logica pentru client, parsarea fisierului de intrare şi generarea fişierului de ieşire.

ATENȚIE! Functia `unsigned int hash_function_servers(void *a)` va fi folosită pentru a genera hash-ul etichetei unui server, iar funcţia `unsigned int hash_function_key(void *a)` va fi folosită pentru a genera hash-ul unui obiect. **Folosiți aceste funcții pentru a calcula pozitiile pe hashring**, dar nu le modificaţi deoarece veţi obţine alte rezultate decât cele aşteptate de checker.

Explicații adiționale

Puteti vizualiza aici [https://docs.google.com/presentation/d/1gdy9Al3KtNzUL2a42d9lbDxnlUfn4P0FUX62FR9OPcs/edit?usp=sharing] fiecare operatie din test1.in.

Punctaj

- 80p teste: **fiecare** test este verificat cu valgrind. Dacă un test are memory leaks, nu va fi punctat.
- 10p coding style
- 10p README
- O tema care nu compilează va primi 0 puncte.

Nu copiați! Toate soluțiile vor fi verificate folosind o unealtă de detectare a plagiatului. În cazul detectării unui astfel de caz, atât plagiatorul cât și autorul original (nu contează cine e) vor primi punctaj 0 pe toate temele! De aceea, vă sfătuim să nu vă lăsați rezolvări ale temelor pe calculatoare partajate (la laborator etc), pe mail/liste de discuții/grupuri etc.

FAQ

Q: Putem modifica scheletul / adăuga funcţii?
A: Da. În schimb, nu puteti modifica antetele functiilor deja existente in load_balancer.h / server.h.

Q: Ce funcţii de hashing putem folosi pentru server?
A: Puteţi folosi orice funcţie doriţi (inclusiv pe cele din laborator).

Q: Cum funcţionează checker-ul?
A: Checker-ul verifică faptul că obiectele adăugate de voi pe servere să respecte proprietăţile aflate în cerinţă. În fişierele ref vor apărea mesaje de forma *"Stored #product_name on server #server_id."* şi *"Retrieved #product_name from server #server_id."* În cazul schimbării funcţiei de hashing pentru servere / obiecte sau al schimbării criteriului de matching dintre un server şi un obiect, vor apărea diferenţe între rezultatul vostru şi cel din fişierul ref.

Q: Putem implementa tema în C++?
A: Nu.

Link-uri utile

https://www.toptal.com/big-data/consistent-hashing [https://www.toptal.com/big-data/consistent-hashing]
https://www.youtube.com/watch?v=zaRkONvyGr8&ab_channel=GauravSe [https://www.youtube.com/watch?v=zaRkONvyGr8&ab_channel=GauravSe]

sd-ca/teme/tema2-2021.txt · Last modified: 2021/04/22 23:25 by andrei_tudor.topala