



## Hashing

Hashing: An ~~alg~~ alg<sup>n</sup> ie. performed to transform an arbitrary piece of data (str., arr, an other data type or struc) into a fixed size value, typically an int.

e.g. in sys. des:- Arbitrary Data → IP address, HTTP request, username,

1

## Need for Hashing

Clients

(C1)

(C2)

(C3)

(C4)

L-B.

LB.

Servers

(S1)

(S2)

(S3)

(S4)

① While servers select strategies such as round robin or random get your work done in normal scenarios, they fail to optimize cache hits in a in-memory cache scenario.

② In Memory Cache

a) Cache hit → no. of times a req. received a cached res. from our sys.  
(faster)

(SSS)

- 6) For this, round robin or random server select strategy don't work as they do not guarantee you that if the 1<sup>st</sup> time a client makes a req. and it gets ~~processed~~ a processed and cached in S1, the 2<sup>nd</sup> time, when client makes the same req., he/she will be redirected to S1 only  
∴ chances of cache hits ↓

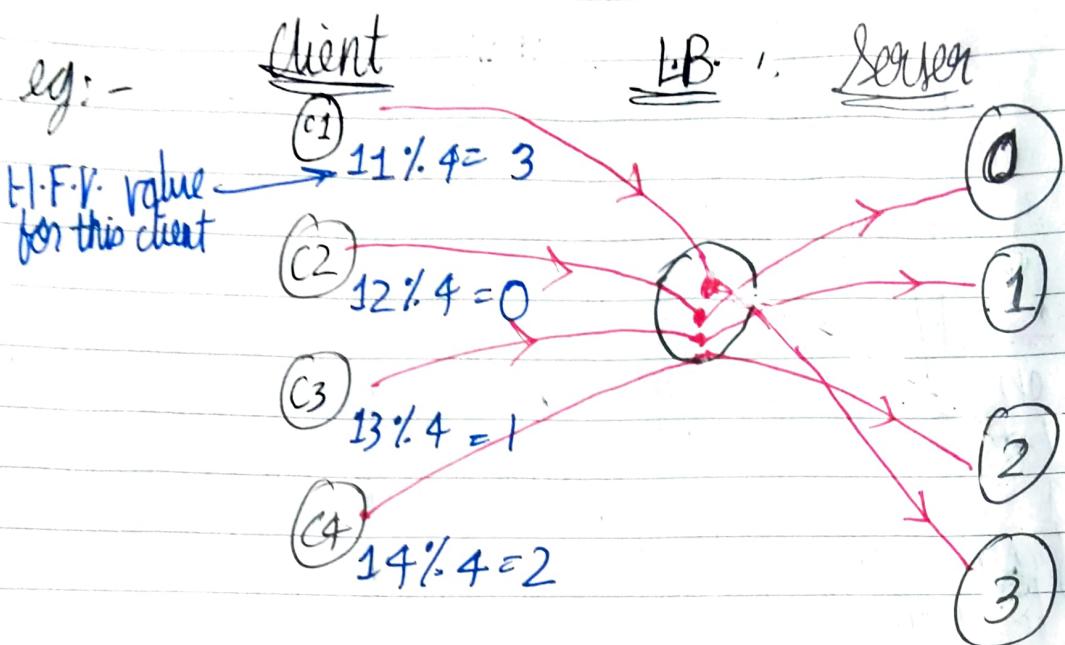
- c) ∴ If your sys. des is st., it relies heavily on in-memory cache in its servers, but your L.B. is receiving req.s from clients following a SSS that doesn't guarantee that reqs. from some client OR same reqs. will be routed to the same server every time, then, your in-memory caching sys. falls apart

- (3) ∴ Hashing will basically take these reqs →
- the 1<sup>st</sup> time req is processed, cached in that server ← based on this value, req is allotted to a server. → hash them to a fixed value
- any subseq. times, the same req. is again sent to L.B. → now for the same req. we get the same hashed fixed value (H-F-V.)
- same server ← same hashed fixed value → so same server so same resp. i.e. the cached resp.

## 2] Simple Hashing $\rightarrow$

$$\text{server allotted to client (SATC)} = (\text{H.F.V.}) \% \text{ (no. of servers)}$$

e.g.:-



- ① Uniformity in Hashing Function (H.F.)  $\rightarrow$  No. of collisions
- i.e. same server allotted to a no. of clients should be minimized.
  - i.e. clients should be distributed across servers equally, uniformly.

NOTE: Typically ~~you~~ you never write your own H.F., You use ~~the~~ pre-made industry-grade H.F.s. such as MD-5 or SHA-1 or Bcrypt

- ② Cache hits  $\uparrow$  for same reqs

- ③ Problem :- Adding or removing ~~one~~ (or any) of servers completely fucks up cache routes (C.R.  $\rightarrow$  a route which a req. must follow to get its matching cache a.k.a. route for cache hits)

$\text{MEM} \rightarrow \text{Memory}$

Date \_\_\_\_\_  
M-T-W-T-F-S

e.g.: - Adding a server to cache, e.g. after in-mem. caches have been created for reqs which already in their resp. SATC. Say server S5 is added now

Client	H·F·V	$n_i = 4$	$n_f = 5$	SATC <sub>i</sub>	SATC <sub>f</sub>
C1	11	4	5	3	1
C2	12	4	5	0	2
C3	13	4	5	1	3
C4	14	4	5	2	4

$n_i$  = initial no. of servers

$\text{SATC}_i$  = initial order (the one with in mem. cache) allotted to a req. from this client

- As visible, no new reqs (i.e. reqs after adding of S5) are going on their cache route.
- ∵ They won't reach server with cached res.
- ∵ Cache hits ↓

### 3] Consistent Hashing

SAP → Server Allotment Procedure

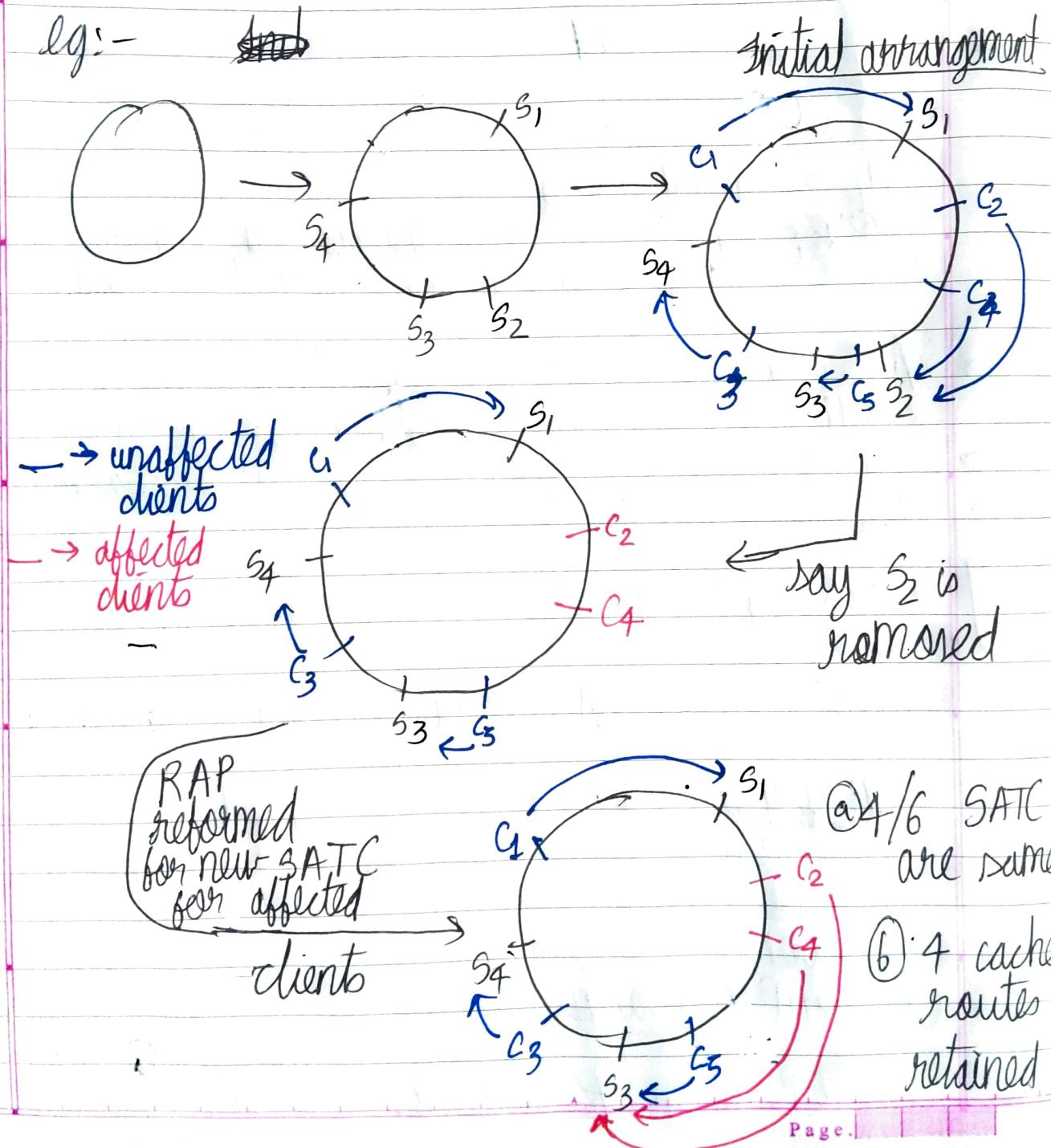
Visualisation: ① Servers are placed on a circle, preferably at equal dist. covering entire circle in equal parts.

- ② Clients are now placed on the same circle
  - ③ SAP → Matching client to its closest server if we traverse circle in  $\frac{1}{2}$  dir<sup>n</sup>
- SAP is performed to get SATC for each client

④ Why circle? :- Cuz it's a shape that is closed

∴ It is easy to just say that on adding or deleting a server, we know the next closest server just for the clients affected and so when we re-perform SAP to get SATC for each client we retain cache routes for all unaffected clients

e.g:- ~~Diagram~~



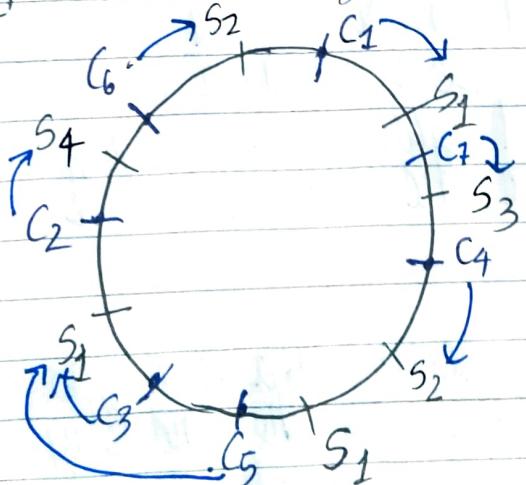
Thus, we retain at least some cache routes irresp. of addin' / removal of server.

high latency

NOTE: ① Also, as u see,  $C_1$  is very far from  $S_1$ . This prob. can be solved by arranging servers at equal dist.s, coverin entire area of wide ~~real world~~ set up servers in geographical loca's s.t. you minimize dist.s b/w all servers and clients mappin to em

② We can traverse circle in P or G dir.  
 Dir doesn't really matter as long as it's fixed.

③ Say if you have a more powerful server i.e. server scaled vertically, just place it on more pts. in circle to ensure more clients are directed to it ~~real world~~ place powerful server/ scale a server vertically, in geographical loca's where no. of clients is high.



$$\text{Power} \rightarrow S_1 > S_2 > S_3 = S_4 = S_5$$

$S_1$  serves 3 clients

$S_2$  serves 2 clients

$S_3, S_4$  serve 1 client

## Rendezvous Hashing:

- i) For every client, H.F. calculates and allocates score for each server (i.e. ranks the servers) (H.R.S.)
- ii) The highest ranking server is chosen as the destination server for our client. A uniform H.F. will ensure that for each client, we're a diff. H.R.S. i.e., a diff. destination server.
- iii) Thus, each server acts as an H.R.S. for same client.
- iv) ~~Removing server~~ → If a server is removed then we go to the client associated with it and we then allot the 2<sup>nd</sup> highest ranking servers for that client as the new destination server for that client.
- v) ~~Adding a server~~ → Re-ranking carried out for clients but new server will be pushed as H.R.S. only in 1 ranking so 1 client's destin. server will change but for other clients' ~~destn.~~ servers won't change so cache routes won't change so reqs. will still keep getting to the server with the in-mem. - cache.

Note: SHA → Secure Hash Algos → "older" of cryptographic H.F.s in the industry. These days, SHA-3 is a popular choice to use in a sys.

```
const serverSet1 = [ 'server1', 'server2', 'server3', 'server4', 'server5', ]  
const serverSet2 = [ 'server1', 'server2', 'server3', 'server4', ]  
  
const username = [ 'username0', 'username1', 'username2', 'username3', ..... 'username9' ]  
  
//SIMPLE HASHING  
function hashString(string) {  
    let hash = 0;  
    if (string.length === 0) return hash;  
    for (let i = 0; i < string.length; i++) {  
        charCode = string.charCodeAt(i);  
        hash = (hash << 5) - hash + charCode  
        hash |= 0  
    }  
    return hash}  
  
//RENDEZVOUS HASHING  
function computeScore(username, server) {  
    const userNameHash = hashString(username);  
    const serverHash = hashString(server);  
    return (userNameHash * 13 + serverHash * 11) % 67}  
  
//SIMPLE HASHING  
function pickServerSimple(username, servers) {  
    const hash = utils.hashString(username);  
    return servers[hash % servers.length]}  
  
//RENDEZVOUS HASHING  
function pickServerRendezvous(username, servers) {  
    let maxServer = null;  
    let maxScore = null;
```

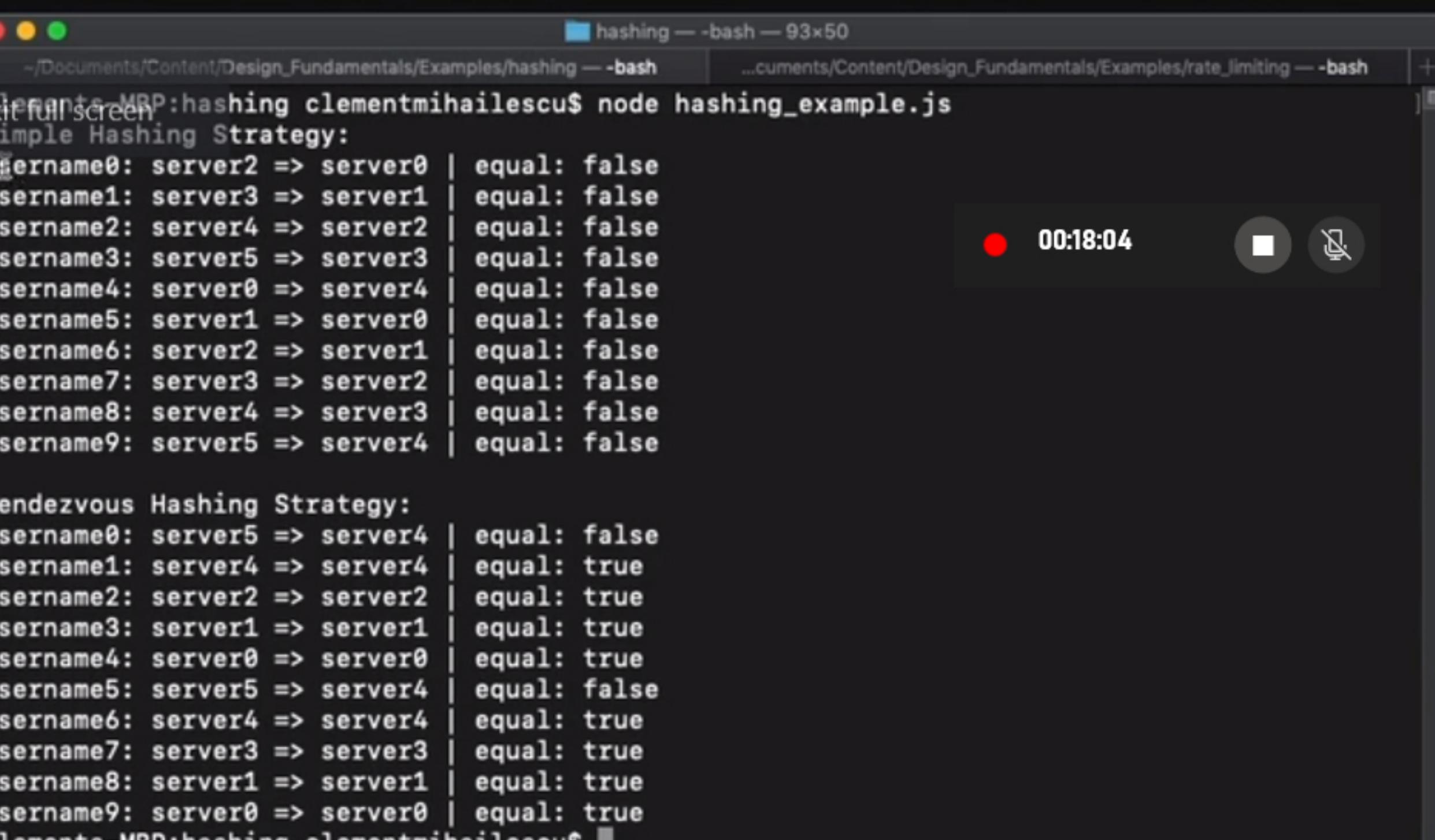
```
for (const server of servers) {  
    const score = utils.computeSCore(username, server);  
    if (maxScore === null || score > maxScore) {  
        maxScore = score;  
        maxServer = server  
    }  
}  
  
return maxServer  
}
```

#### //SIMPLE HASHING

```
console.log('Simple Hashing Strategy')  
for (const username of usernames) {  
    const server1 = pickServerSimple(username, serverSet1);  
    const server2 = pickServerSimple(username, serverSet2);  
    const serversAreEqual = server1 == server2  
}
```

#### //RENDEZVOUS HASHING

```
console.log('Rendezvous Hashing Strategy')  
for (const username of usernames) {  
    const server1 = pickServerRendezvous(username, serverSet1);  
    const server2 = pickServerRendezvous(username, serverSet2);  
    const serversAreEqual = server1 == server2  
}
```



```
hashing_example.js — hashing
JS hashing_example.js × JS hashing_utils.js
JS hashing_example.js > [e] server1
...
36
37  function pickServerRendezvous(username, servers) {
38    let maxServer = null;
39    let maxScore = null;
40    for (const server of servers) {
41      const score = utils.computeScore(username, server);
42      if (maxScore === null || score > maxScore) {
43        maxScore = score;
44        maxServer = server;
45      }
46    }
47    return maxServer;
48  }
49
50  console.log('Simple Hashing Strategy:');
51  for (const username of usernames) {
52    const server1 = pickServerSimple(username, serverSet1);
53    const server2 = pickServerSimple(username, serverSet2);
54    const serversAreEqual = server1 === server2;
55    console.log(` ${username}: ${server1} => ${server2} | equal: ${serversAreEqual}`);
56  }
57
58  console.log('\nRendezvous Hashing Strategy:');
59  for (const username of usernames) {
60    const server1 = pickServerRendezvous(username, serverSet1);
61    const server2 = pickServerRendezvous(username, serverSet2);
62    const serversAreEqual = server1 === server2;
63    console.log(` ${username}: ${server1} => ${server2} | equal: ${serversAreEqual}`);
64  }
Press Esc to exit full screen
```

Like here, you're thinking, okay well,