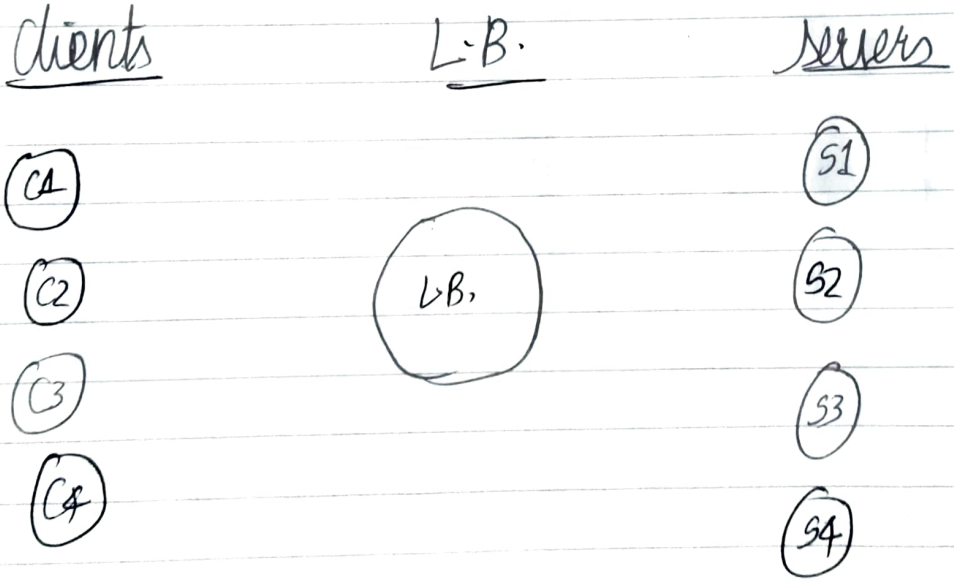
Hashing

Hashing: An algo ie. performed to transform an arbitrary piece of data (str., arr, or other data type or struc) into a fixed size value, typically an int.

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

1 Need for Hashing



(1) While server selec" 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 hits → no of times a req. recieves a cached res. from our sys. (faster)

(SSS)

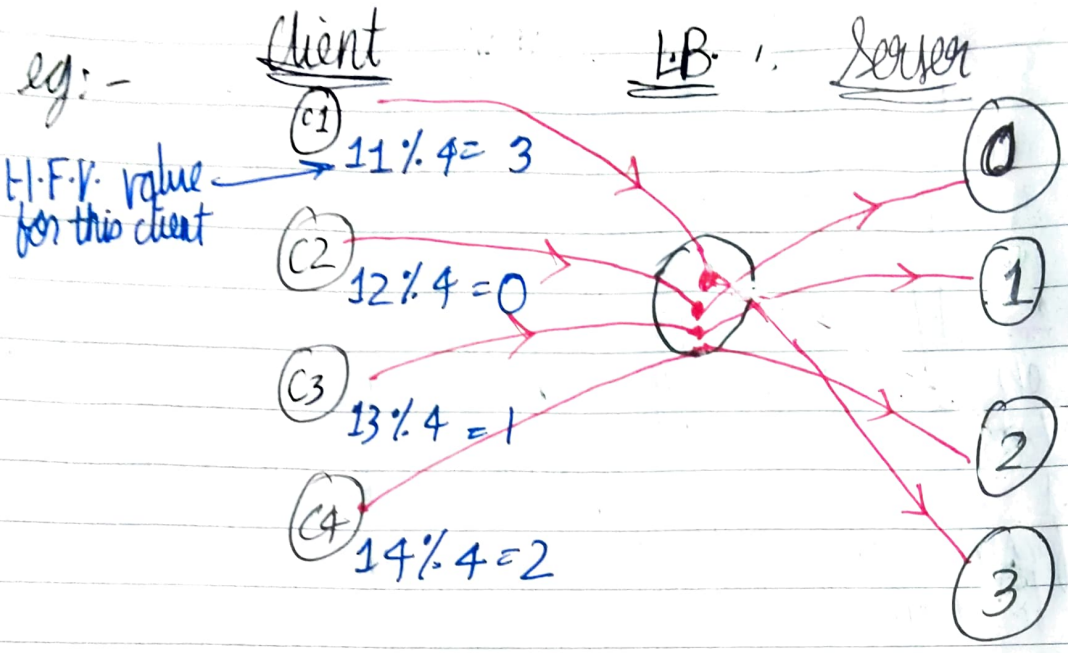
6) For this round robin or random server selection strategy don't work as they do not guarantee you that if the 1st time a client makes a req. and it gets processed and cached in S1, the 2nd time, when client makes the same req., he/she will be redirected to S1 only
∴ Chances of cache hits ↓

2) ∴ If you sys. des is s.t., it relies heavily on in-memory cache in its servers, but your L.B. is rerouting req.s from client, following a SSS that doesn't guarantee that: reqs from same 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 → hash them to a fixed value → based on this value, req is allotted a server. → the 1st time req is processed, cached in that server → any subseq. times, the same req. is again sent to L.B. → now for the same req. we get the same hashed fixed rate (H.F.V.) → same hashed fixed value → so same server → same server so same resp. i.e. the cached resp.

2] Simple Hashing → server allotted to client (SATC) = (H.F.V.) % (no. of servers)

eg:-



(1) Uniformity in Hashin f^n (H.F.) → No. of collisions i.e. same server allotted to a no. of clients should be minimized.
i.e. clients should be distri.ed across servers equally, uniformly

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

(2) Cache hits ↑ for same reqs.

(3) Problems:- Addin or removin (or dyin) of servers completely fucks up cache routes (C.R. → a route which a req. must follow to get its matchin cache a.k.a. route for cache hits)

mem → memory

eg:- Addn a server to curr. eg. 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.	$n_i = 4$	$n_f = 5$	$SATC_i$	$SATC_f$
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
$SATC_i$ = initial server (the one with in mem. cache) alotted to a req. from this client

→ As visible, no new reqs (i.e. reqs after addi^n of S5) are goin on their cache route
→ ∴ They won't reach server with cached res.
→ ∴ Cache hits ↓

3] Consistent Hashing

SAP → Server Allotment Procedure

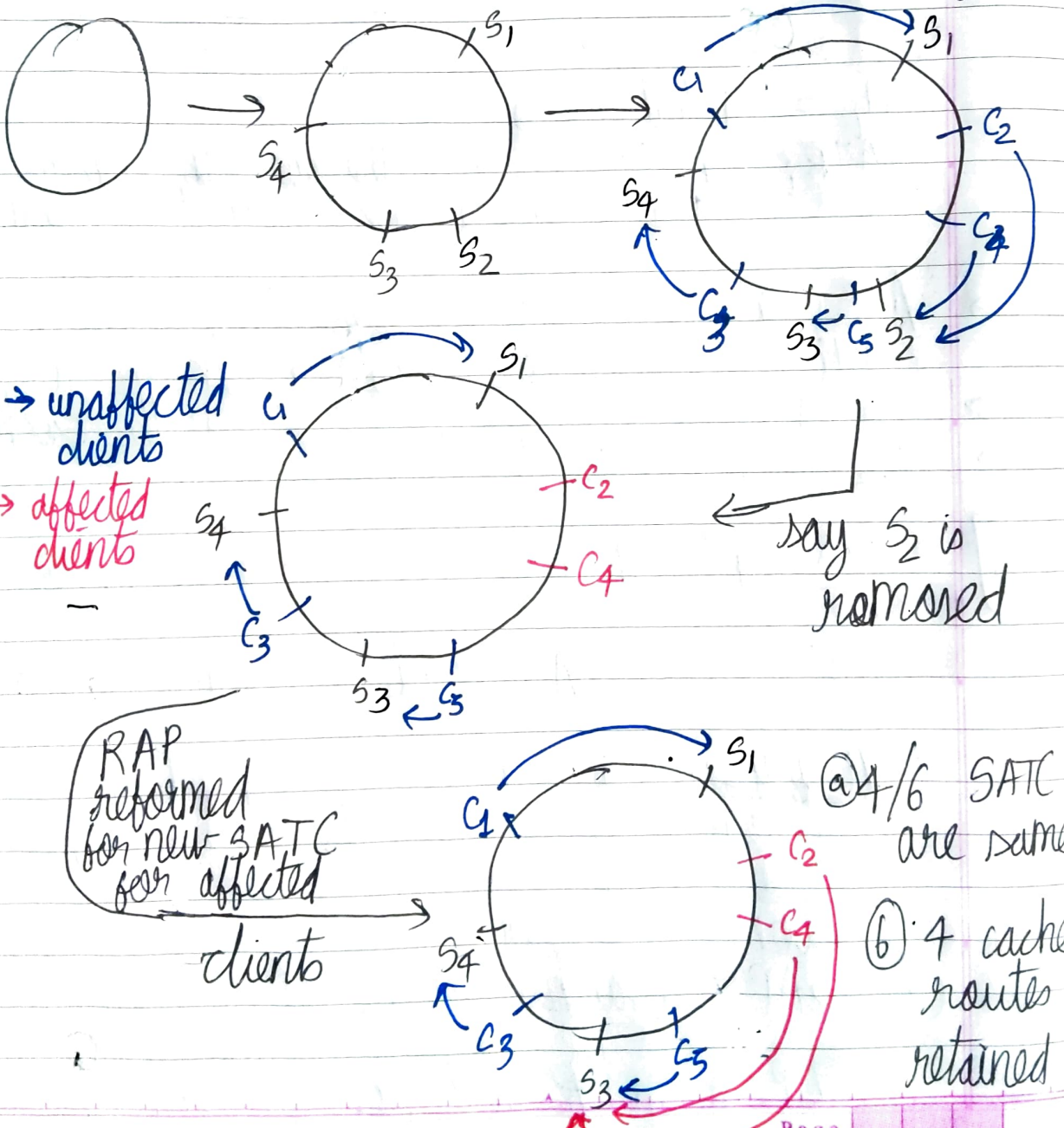
Visualiss^n : ① Servers are placed on a circle (preferably at equal dist. coverin entire circle in equal parts.
② Clients are now placed on the same circle
③ SAP → Matchin client to its closest server if we traverse circle in ? dir^n
SAP is performed to get SATC for each client

Date:
M T W T F S S

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

∴ It is easy to just say that on addin 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

eg:-



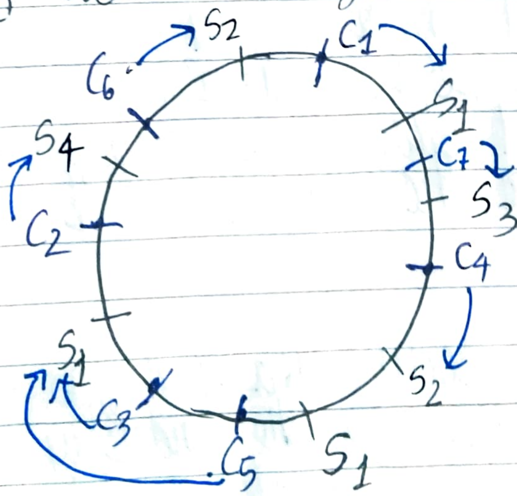
Thus, we retain altleast some cache routes irresp. of addiⁿ / removal of ^ server.

NOTE: ① Also, as visib, C_1 is very far from S_1 (high latency).
This prob. can be solved by arrangin servers at equal dist.s, coverin entire area of circle —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 ↻ or ↺ dirⁿ. Dirⁿ 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.

Power → $S_1 > S_2 > S_3 = S_4 = S_5$



S_1 serves 3 clients
S_2 serves 2 clients
S_3, S_4 serve 1 client

Date
M T W T F S S

Rendezvous Hashing:

i) For every client, H.F. calculates and allots score for each server (i.e. ranks the servers)

(H.R.S.)

ii) The highest ranking server is chosen as the destina" server for our client. A uniform H.F. will ensure that for each client, we'll a diff. H.R.S. i.e., a diff. destina" 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 2nd highest ranking server for that client as the new destina" 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 destn. 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 → collec" of cryptographic H.F.s in the industry. These days, SHA-3 is a popular choice to use in a sys.

Page

```
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 = hashStrinng(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
  }
```