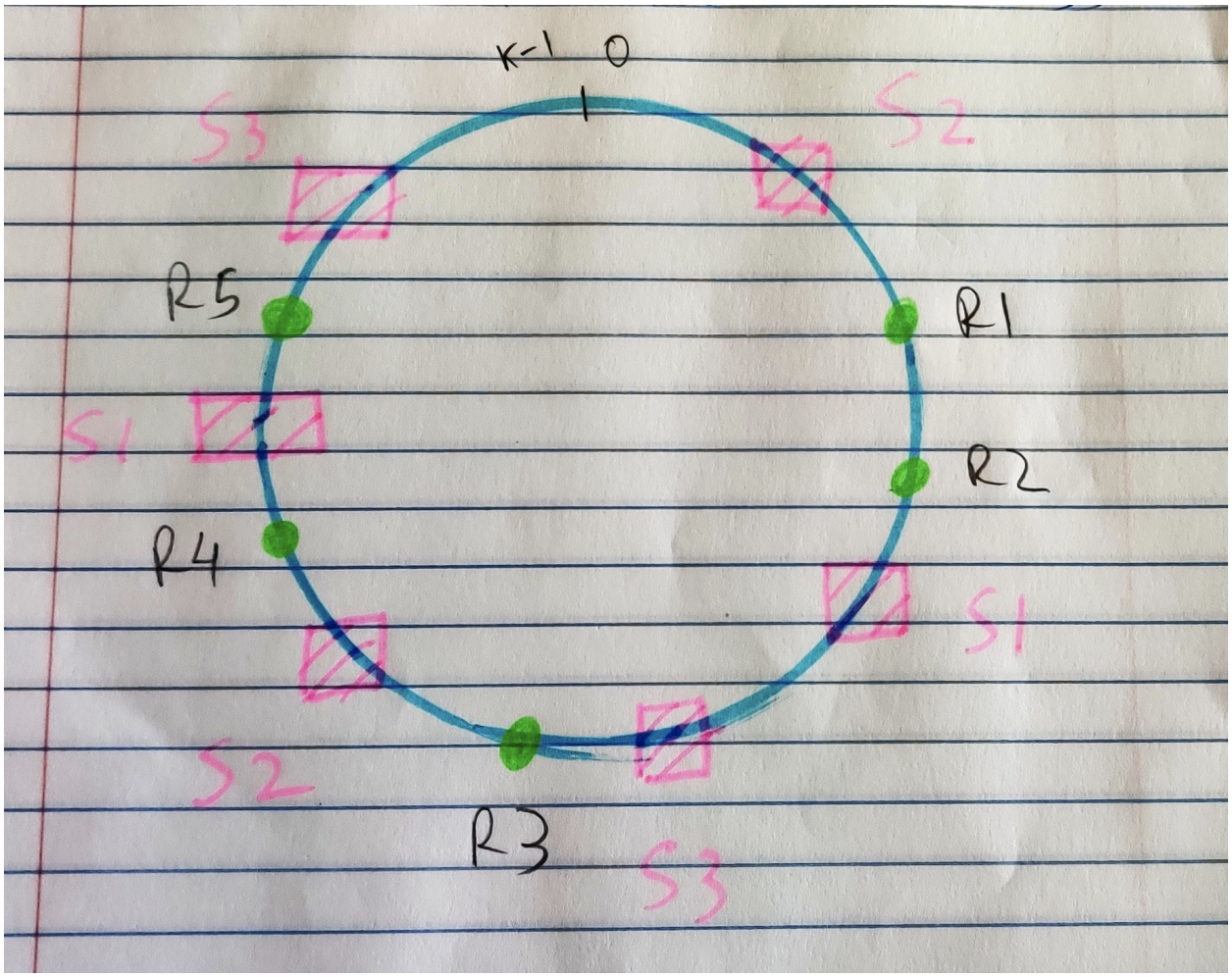
Consistent hashing with Bounded Loads

Consistent Hashing Overview:



- As we all know from Lab 5, a typical consistent hashing algorithm involves a ring with K buckets numbered from 0 to K - 1.
- The idea is that each Server we have is mapped to a point on the ring with the help of a hash function (typically less expensive one's like MurmurHash, MetroHash, Ketama, etc.).
- When a request comes in, we hash the content of the request (like request id, session id, etc.) and map it on the ring.

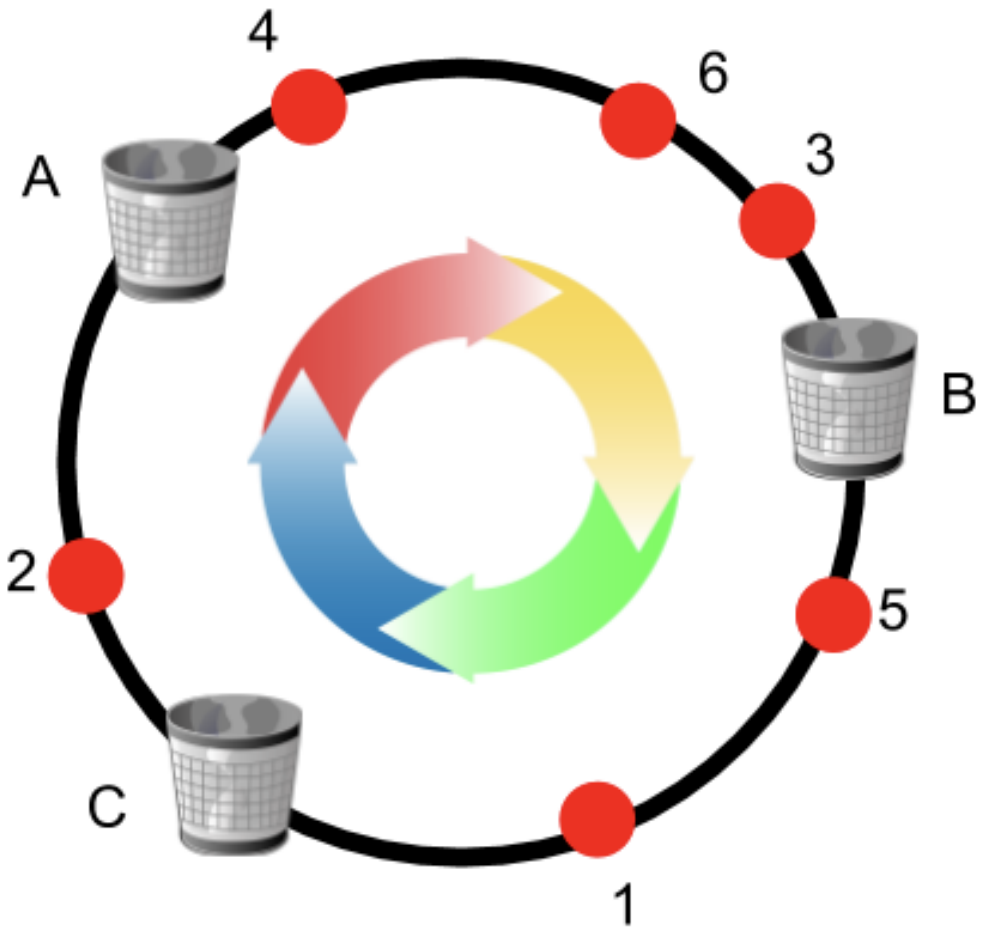
- After mapping the request, we move forward in the ring until we find a server and this server will serve our request.
- The benefit of this approach is that most requests will be mapped to the same server when a server is removed or added.
- In practice, each server is mapped to multiple points on the ring typically using algorithms like Chord.
- These extra points are called Virtual Nodes.
- Having multiple points reduces the load variance among servers.

Drawbacks of Consistent Hashing:

- One drawback of consistent hashing is that because of it's property to map to the same server for the same request, if some content is much more popular than other, the server might get overloaded resulting into a drop in traffic.

Consistent Hashing with Bounded Load Algorithm:

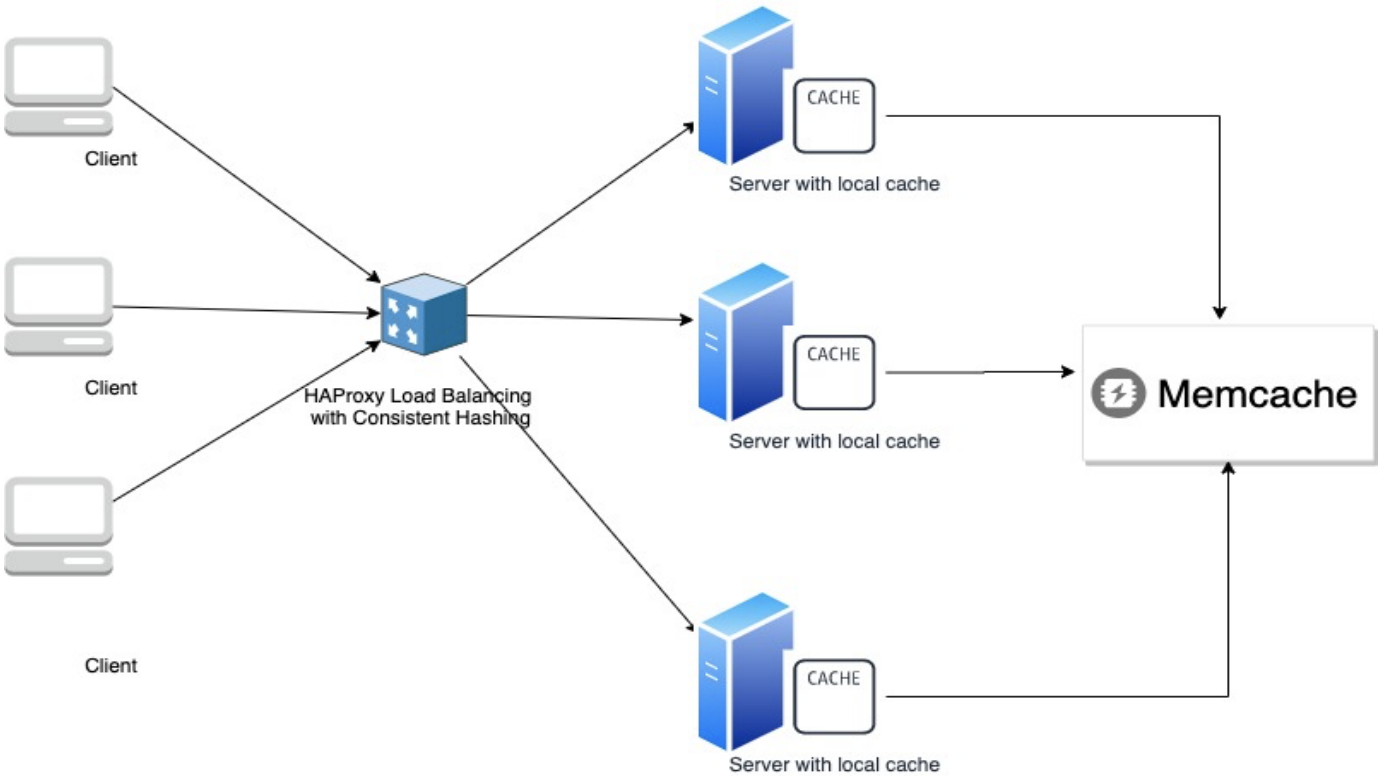
- The idea to define a bound on the servers such that none of them get overloaded was first conceived at google.
- In order to avoid overloading of the server, we can define a balancing factor, c, which will define the maximum load a server will get.
- For example, if c = 1.25, no server will have loads greater than 1.25 times the average load.
- Let's understand this by taking an example.



- In the above figure, consider buckets A, B and C as our Servers and the red balls as our requests.
- Also, let us define the maximum load each server can take, say, 2 requests.
- Now, according to the bounded load algorithm, requests 6 and 3 will go clockwise in the ring and it will be served by server B.
- Similarly, requests 1 and 5 will be served by C and request 2 by A.
- Now, for request 4, when it goes to server B, B is already at its capacity, so it will move forward in the ring to the next server, i.e. C, but server C is also at its max capacity, hence it will go to server A, which hasn't reached its capacity and the request will be served by A.
- Hence this will guarantee that,
    - No server will get overloaded by more than a factor of c,
    - The distribution of requests is the same as consistent hashing as long as the servers aren't overloaded.
    - In case a server is overloaded, our request will consistently be mapped to the fallback server or our "second choice" of server.

Improving Load Balancing at Vimeo:

- Vimeo was one of the first companies to implement this Algorithm in the HAProxy Code base.
- On a high level, Vimeo's architecture is something like this,



- They use HAProxy Software as Load Balancing service and consistent hashing as their load balancing algorithm.
- When a request for a particular video id comes in, it will be redirected to the same server using consistent hashing.
- Hence, for the same video, the request will be redirected to the same server. So it makes sense to cache the video in the local cache of the server.
- However, they observed that the % of local cache hit was just around 50 % because they used the least connection policy along with consistent hashing to make sure that the server doesn't get overloaded.
- So to overcome this problem, they added a second layer of global caching service in the form of memcache.

- Now if the request for the same video comes in, it will first check in the local cache, and if not found, it will search the global cache.
- This improved the performance of the system.
- However, they still needed to figure out a way to optimally use the resources.
- So, after looking around, they found this paper (https://arxiv.org/abs/1608.01350) and decided to implement it in the HAProxy source code.

HAProxy Source Code changes:

- Below is the function which will find the next server in the ring which hasn't exceeded the defined capacity and return that server's object.

```c
struct server *chash_get_server_hash(struct proxy *p, unsigned int hash)
{
    struct eb32_node *node, *stop;
    struct server *s;
    struct eb_root *root;

    if (p->srv_act)
        root = &p->lbprm.chash.act;
    else if (p->lbprm.fbck)
        return p->lbprm.fbck;
    else if (p->srv_bck)
        root = &p->lbprm.chash.bck;
    else
        return NULL;

    /* find the next node */
    stop = node = eb32_lookup_ge(root, hash);
    if (!node)
        node = eb32_first(root);
    if (!node)
        return NULL; /* tree is empty */

    chash_update_server_capacities(p);

    s = eb32_entry(node, struct tree_occ, node)->server;

//  send_log(p, LOG_WARNING, "server %s: %d conns, %d cap\n", s->id, s->served + s->nbpend, s->chf_cap);

    while (s->served + s->nbpend >= s->chf_cap) {
        node = eb32_next(node);
        if (!node)
            node = eb32_first(root);
        s = eb32_entry(node, struct tree_occ, node)->server;
        if (node == stop) {
            send_log(p, LOG_ERR, "chash_get_server_hash went all the way around");
            break;
        }
    }

    return s;
}
```

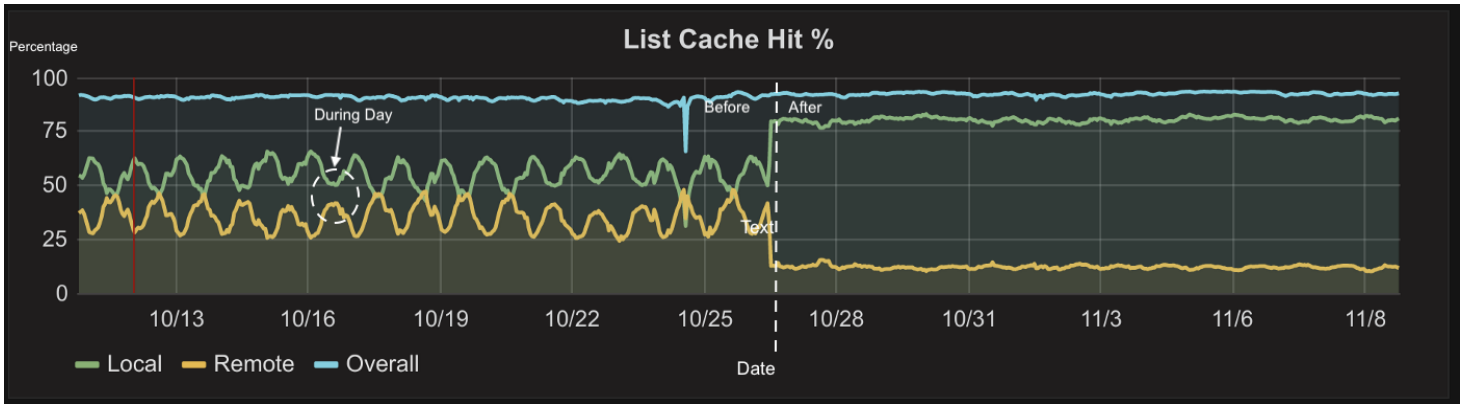
- If we look at the highlighted area, we will find the logic of checking the server capacity.
- Here is the link to the file : https://github.com/arodland/haproxy/blob/master/src/lb_chash.c

```
315     while (s->served + s->nbpend >= s->chf_cap) {
316         node = eb32_next(node);
317         if (!node)
318             node = eb32_first(root);
319         s = eb32_entry(node, struct tree_occ, node)->server;
320         if (node == stop) {
321             send_log(p, LOG_ERR, "chash_get_server_hash went all the way around");
322             break;
323         }
324     }
325
326     return s;
```

- As we can see here, the while loop will check whether the number of current connections + number of pending connections of server object, s, does not exceed the chf_cap(consistent hashing and forwarding capacity) of our server.
- If it does, it will get the next server in the eb_tree(elastic binary tree), and check for the same condition again.
- If it finds any server which does not exceed capacity, it will break and our code will return the server object.
- If not, we will continue to find a server until we reach the server from which we started the entire process.
- If that is the case, we break from the loop and return null.

Benchmarks:

- The changes made in the HAProxy codebase were pushed into production on 26th October 2016.
- From the image below, we can clearly see the improvement in the % of local cache hits.



- The daily variation before was caused by autoscaling: during the day, there's more traffic, so they started more servers to handle it, and fewer requests could be served by local cache.
- At night, there's less traffic, so they shut servers down, and the local cache performance went up somewhat.
- After switching to the bounded-load algorithm, a much bigger fraction of requests hit local cache, regardless of how many servers were running.

- As a result of this, the typical bandwidth used by each Memcached server was 400-500 Mbps, maximizing at 900 Mbps in peak hours.
- After implementing the algorithm, the servers stay consistently below 100 Mbps.
- As a result,
    - Now a much smaller number of requests rely on memcache servers.

    - Hence, they can now serve more number of requests with the same amount of cache.
    - In addition to this, if the memcache server ever goes down, it will have minimal effect on average performance of the system.

Closing remarks:

- This implementation was later refactored and is Generally Available starting HAProxy version 1.7.0.
- You can find it here : https://github.com/haproxy/haproxy/blob/master/src/lb_chash.c

Youtube Link of my presentation :

https://www.youtube.com/watch?v=aUF5erIkBi8

References:

1. https://ai.googleblog.com/2017/04/consistent-hashing-with-bounded-loads.html
2. https://arxiv.org/abs/1608.01350
3. https://medium.com/@dgryski/consistent-hashing-algorithmic-tradeoffs-ef6b8e2fcae8
4. https://netflixtechblog.com/distributing-content-to-open-connect-3e3e391d4dc9
5. https://medium.com/vimeo-engineering-blog/improving-load-balancing-with-a-new-consistent-hashing-algorithm-9f1bd75709ed
6. https://medium.com/system-design-blog/consistent-hashing-b9134c8a9062
7. Source Code : https://github.com/arodland/haproxy/