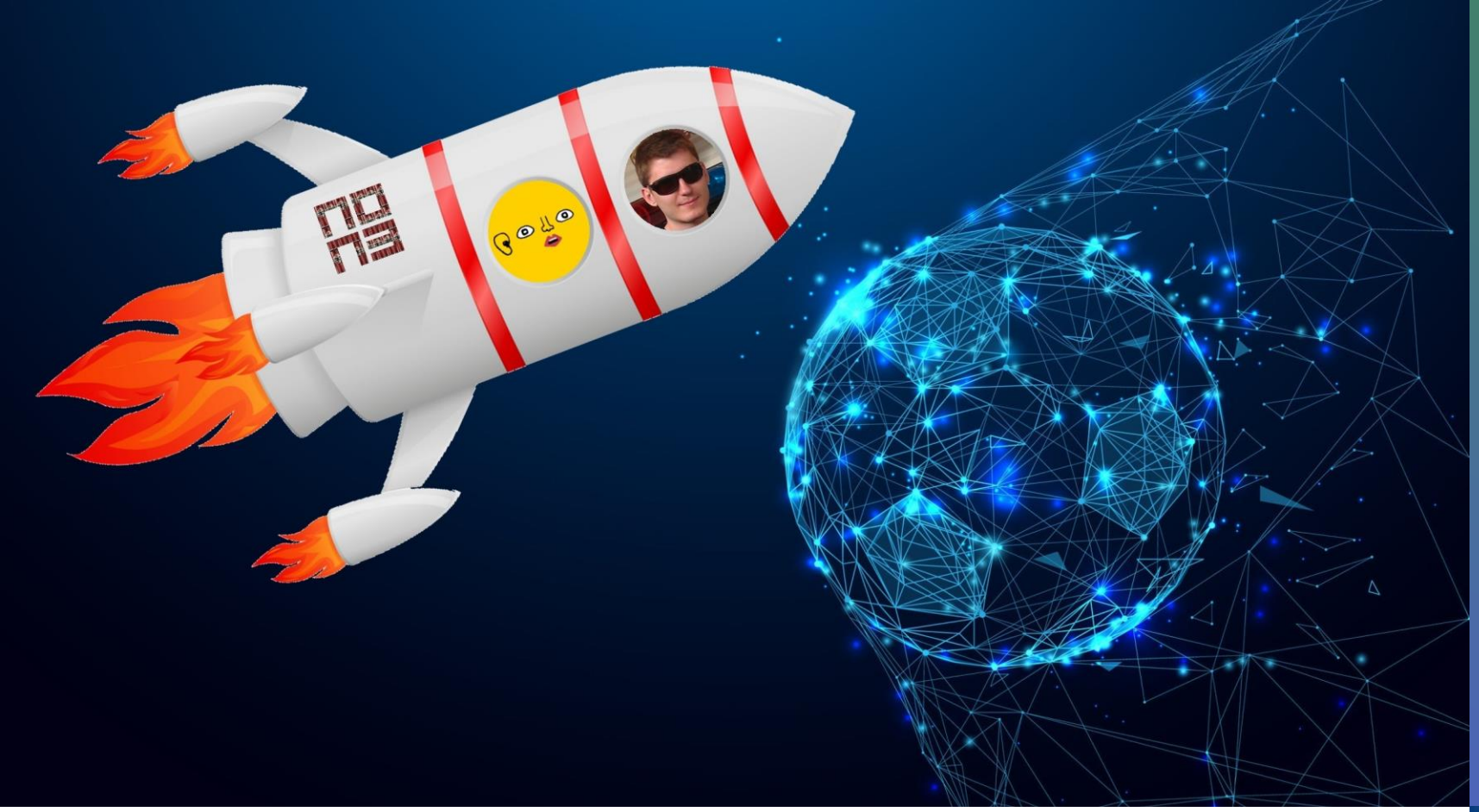
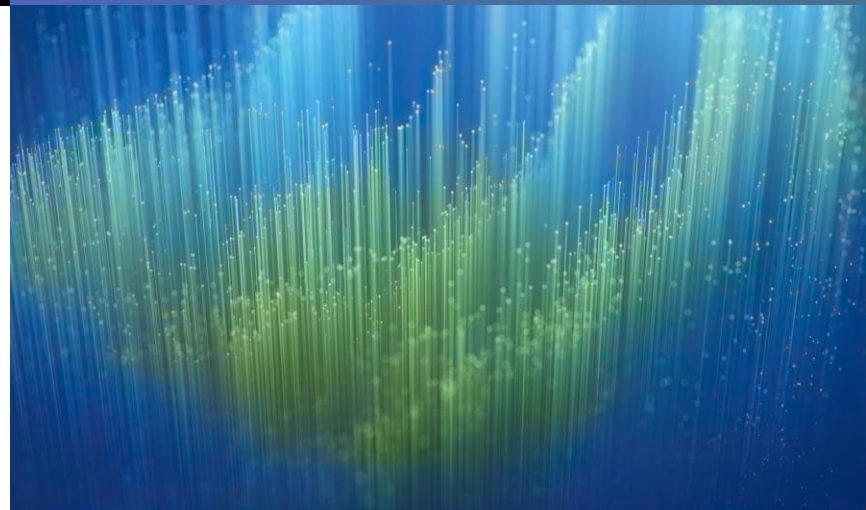


ПОДГОТОВИЛА
КРИГЕР ВИКТОРИЯ,
18205

КОНСИСТЕНТНОЕ ХЭШИРОВАНИЕ С ОГРАНИЧЕНИЯМИ. РЕАЛИЗАЦИЯ В HAPROXY



ОБЩИЕ СВЕДЕНИЯ О ХЭШИРОВАНИИ

- Общую информацию о консистентном хэшировании и консистентном хэшировании с ограничениями можно найти в этой статье: Vimeo – Improving load balancing with a new consistent-hashing algorithm - https://medium.com/vimeo-engineering-blog/improving-load-balancing-with-a-new-consistent-hashing-algorithm-9f1bd75709ed
- Это оригинал статьи, здесь будет представлен ее частичный вольный перевод

ОБЫЧНОЕ ХЭШИРОВАНИЕ

- Как именно происходит выбор бэкенда (сервера) для перенаправления запроса при обычном хэшировании? Берется хэш от URL из HTTP запроса, поступившего на фронтенд HAProxy (в случае Vimeo, это часть URL, содержащая ID видео, но в общем случае это хэш от URL), целочисленно делится на количество доступных серверов, в результате получаем индекс сервера, на который будет направлен запрос.
- Такой подход работает нормально до тех пор, пока количество бэкендов не начнет изменяться (удаляются/добавляются серверы).
- В таком случае, мы не сможем эффективно использовать данные из кэшей серверов. В случае Vimeo, на определенных серверах кэшируются данные об определенных видео (в Vimeo используется динамическая подкачка видео, основанная на подкачке сегментов видео, индексы байтов начала сегментов как раз и находятся в кэшах на серверах), таким образом запросы к одним и тем же видео идут на одни и те же сервера.
- Однако при использовании обычного хэширования, при изменении количества серверов, мы каждый раз будем обращаться к разным серверам, что неэффективно.

КОНСИСТЕНТНОЕ ХЭШИРОВАНИЕ

- Поэтому было решено использовать консистентное хэширование.
- При консистентном хэшировании каждому серверу сопоставляется свой собственный хэш (берется от его имени или ID), далее берется хэш от URL из HTTP-запроса (равно как и в случае обычного хэширования), и для хэша запроса определяется сервер с "ближайшим" хэшем (хэш = число, а значит сервер с наиболее близким числом к хэшу запроса). На него и направляется запрос.
- Это решает проблему эффективного использования кэшей серверов, так как при добавлении/удалении сервера, только для небольшой части запросов изменится свой ближайший сервер, остальные запросы продолжат поступать на те же самые серверы.

КОНСИСТЕНТНОЕ ХЭШИРОВАНИЕ С ОГРАНИЧЕНИЯМИ

- Однако, при консистентном хэшировании возникает проблема неравномерной загруженности серверов: если одно видео очень популярно, запросов на один сервер станет слишком много. Для решения этой проблемы было использовано консистентное хэширование с ограничениями: ссылка на оригинальную статью авторов алгоритма https://arxiv.org/abs/1608.01350
- Идея консистентного хэширования с ограничениями такова: подбирается балансирующий фактор *c* (некая константа большая 1) , определяющая максимально допустимую долю имбаланса между серверами. Например, при c = 1,25, ни один сервер не должен быть загружен более, чем на 125% от средней нагрузки. Авторы отмечают, что c между 1,25 и 2 наиболее эффективно на практике.

КОНСИСТЕНТНОЕ ХЭШИРОВАНИЕ С ОГРАНИЧЕНИЯМИ

- При каждом новом поступившем запросе, считается средняя нагрузка = avg = количество уже поступивших запросов, включая только что поступивший, деленное на количество доступных серверов; t = avg * c это целевая нагрузка на сервер. Далее для каждого сервера определяется нагрузка - $[t]$ или $\lceil t \rceil$, и общая нагрузка это $[cm]$. Соответственно, максимальная нагрузка на каждый сервер составляет $[cm/n]$ (n - количество доступных серверов).
- Как происходит выбор сервера для перенаправления запроса? Точно так же, имеем два подсчитанных хэша (сервера и запроса), находим ближайший по хэшу сервер, если он еще не загружен полностью, отправляем запрос на него, в противном случае - запрос отправляется на следующий сервер и так, пока не найдется еще не заполненный (гарантируется, что он найдется).

ДЕТАЛИ РЕАЛИЗАЦИИ КОНСИСТЕНТНОГО ХЭШИРОВАНИЯ С ОГРАНИЧЕНИЯМИ В HAPROXY

- Разберемся, как именно реализовано консистентное хэширование в коде HAProxy. Ссылка на репозиторий: https://github.com/haproxy/haproxy

- Бэкенды (серверы) представляются в виде бинарного дерева, а именно Elastic Binary Tree. Общая идея ebtree: https://github.com/haproxy/haproxy/blob/master/include/import/ebtree.h
- Эта структура поддерживает все стандартные операции: поиск по ключу, добавление ноды, удаление ноды, следующая нода и т.д.

EBTREE

Базовая структура ноды дерева struct eb_node:

```
struct eb_node {
    struct eb_root branches; /* branches, must be at the beginning */
    eb_troot_t    *node_p;  /* link node's parent */
    eb_troot_t    *leaf_p;  /* leaf node's parent */
    short int      bit;     /* link's bit position. */
    short unsigned int pfx; /* data prefix length, always related to leaf */
} __attribute__((packed));
```

- На деле используются 32/64-битные версии дерева (в зависимости от ключа key: может быть unsigned int u32, или unsigned long long u64)

```
struct eb32_node {
    struct eb_node node; /* the tree node, must be at the beginning */
    MAYBE_ALIGN(sizeof(u32));
    u32 key;
} ALIGNED(sizeof(void*));
```

- https://github.com/haproxy/haproxy/blob/master/include/import/eb64tree.h
- https://github.com/haproxy/haproxy/blob/master/include/import/eb32tree.h

ХЭШ СЕРВЕРА

- Как было указано ранее, вначале подсчитывается хэш URL из запроса и хэш сервера. Для того, чтобы определить, как подсчитывается хэш сервера, вначале рассмотрим, как сервер представляется в HAProxy.
- Сервер представляет собой структуру struct server: https://github.com/haproxy/haproxy/blob/master/include/haproxy/server-t.h
- У нее очень много полей, нас интересует struct tree_occ *lb_nodes и int puid (/* proxy-unique server ID, used for SNMP, and "first" LB algo */).
- Структура struct tree_occ :

```
/* A tree occurrence is a descriptor of a place in a tree, with a pointer back to the server itself. */

struct tree_occ {

   struct server *server;

   struct eb32_node node;

};
```

ХЭШ СЕРВЕРА

- Файл https://github.com/haproxy/haproxy/blob/master/src/lb_chash.c представляет собой имплементацию консистентного хэширования в HAProxy. В нем находим метод chash_init_server_tree:

```
for (node = 0; node < srv->lb_nodes_tot; node++){
        srv->lb_nodes[node].server = srv;
        srv->lb_nodes[node].node.key = full_hash(srv->puid * SRV_EWGHT_RANGE + node);
    }
```

- Из кода видно, что для каждой ноды дерева подсчитывается хэш функцией full_hash на основании айди сервера и константы SRV_EWGHT_RANGE (определена в https://github.com/haproxy/haproxy/blob/master/include/haproxy/server-t.h) и помещается в поле key :

```
#define SRV_EWGHT_RANGE (SRV_UWGHT_RANGE * BE_WEIGHT_SCALE), где
#define SRV_UWGHT_RANGE 256
#define BE_WEIGHT_SCALE 16 (определена
в https://github.com/haproxy/haproxy/blob/master/include/haproxy/backend-t.h)
```

ХЭШ СЕРВЕРА

- Функция full_hash представляет собой (определена в https://github.com/haproxy/haproxy/blob/master/include/haproxy/intops.h и преобразует один int к другому):

```
static inline unsigned int __full_hash(unsigned int a)
{
    a = (a+0x7ed55d16) + (a<<12);
    a = (a^0xc761c23c) ^ (a>>19);
    a = (a+0x165667b1) + (a<<5);
    a = (a+0xd3a2646c) ^ (a<<9);
    a = (a+0xfd7046c5) + (a<<3);
    a = (a^0xb55a4f09) ^ (a>>16);

    /* ensure values are better spread all around the tree by multiplying by a large prime close to 3/4 of the tree. */
    return a * 3221225473U;
}
```

ХЭШ URL

- Теперь определим, как подсчитывается хэш URL из запроса.
- Обратимся к файлу https://github.com/haproxy/haproxy/blob/master/src/backend.c, найдем метод gen_hash. Его синтаксис достаточно прозрачен, выбирается один из алгоритмов хэширования. По умолчанию указан SDBM алгоритм. Его реализацию можно найти в файле: https://github.com/haproxy/haproxy/blob/master/src/hash.c:
- Выбор алгоритма хэширования может варьироваться, с мнением разработчиков на этот счет можно ознакомиться в документации: https://github.com/haproxy/haproxy/blob/master/doc/internals/hashing.txt

```
unsigned int hash_sdbm(const void
*input, int len)
{
const unsigned char *key = input;
unsigned int hash = 0;
int c;

while (len--){
c = *key++;
hash = c + (hash << 6) + (hash << 16) -
hash;
}

return hash;
}
```

ХЭШ URL

- Непосредственно метод, который по хэшу URL подбирает подходящий сервер, располагается в файле https://github.com/haproxy/haproxy/blob/master/src/backend.c - это метод get_server_ph:

```
static struct server *get_server_ph(struct proxy *px, const char *uri, int uri_len, const struct server *avoid) {

....

hash = gen_hash(px, start, (end - start)); /* тут генерируем хэш URL */

....

return chash_get_server_hash(px, hash, avoid); /* тут по хэшу URL находим подходящий сервер */

}
```

- Метод chash_get_server_hash располагается в файле https://github.com/haproxy/haproxy/blob/master/src/lb_chash.c:

```
/* This function returns the running server from the CHASH tree, which is at the closest distance from the value of <hash>. Doing so ensures that even with a well imbalanced hash, if some servers are close to each other, they will still both receive traffic... */

struct server *chash_get_server_hash(struct proxy *p, unsigned int hash, const struct server *avoid)

{

...

next = eb32_lookup_ge(root, hash); /* находим ноду в дереве (сервер), хэш которого больше или равен, чем хэш URL */ (метод определен в https://github.com/haproxy/haproxy/blob/master/src/eb32tree.c)

…

prev = eb32_prev(next); /* также находим предшествующую найденной ноду (метод определен там же) */

…

/* Смотрим на расстояние до каждой из двух нод, выбираем ту, до которой расстояние минимально */

dp = hash - prev->key;

dn = next->key - hash;

if (dp <= dn) {

    next = prev;

    nsrv = psrv;

...

}
```

Продолжение на след. слайде

```
struct server *chash_get_server_hash(struct proxy *p, unsigned int hash, const struct server *avoid)
{
…
/* Начинаем итерироваться по дереву и смотреть, подходит ли очередной сервер нам: то есть не загружен ли он (проверяется методом chash_server_is_eligible, приведен на следующем слайде, определен в том же файле, где и сам метод chash_get_server_hash). Останавливаемся на первом не загруженном */
while (nsrv == avoid || (p->lbprm.hash_balance_factor && !chash_server_is_eligible(nsrv))){
        next = eb32_next(next);
        if (!next){
            next = eb32_first(root);
            if (++loop > 1) // protection against accidental loop
                break;
        }
        nsrv = eb32_entry(next, struct tree_occ, node)->server;
    }
…
return nsrv;
}
```

```
/* This function implements the "Consistent Hashing with Bounded Loads" algorithm of Mirrokni, Thorup, and Zadimoghaddam (arxiv:1608.01350), adapted for use with unequal server weight
int chash_server_is_eligible(struct server *s)
{
    /* The total number of slots to allocate is the total number of outstanding requests (including the one we're about to make) times the load-balance-factor, rounded up. */
    unsigned tot_slots = ((s->proxy->served + 1) * s->proxy->lbprm.hash_balance_factor + 99) / 100;
    unsigned slots_per_weight = tot_slots / s->proxy->lbprm.tot_weight;
    unsigned remainder = tot_slots % s->proxy->lbprm.tot_weight;

    /* Allocate a whole number of slots per weight unit... */
    unsigned slots = s->cur_eweight * slots_per_weight;

    /* And then distribute the rest among servers proportionally to their weight. */
    slots += ((s->cumulative_weight + s->cur_eweight) * remainder) / s->proxy->lbprm.tot_weight
        - (s->cumulative_weight * remainder) / s->proxy->lbprm.tot_weight;

    /* But never leave a server with 0. */
    if (slots == 0)
        slots = 1;

    return s->served < slots;
}
```

Функциональность довольно прозрачна и понятна из комментариев (в точности реализует идею алгоритма, расписанную на слайдах выше).

Служебная структура lbprm определена в файле https://github.com/haproxy/haproxy/blob/master/include/haproxy/backend-t.h, содержит параметры для балансировки нагрузки