ANDREW LAVERS

zendesk

# Redis is not just a cache

CONFOO MONTREAL 2020

FEB 28, 2020

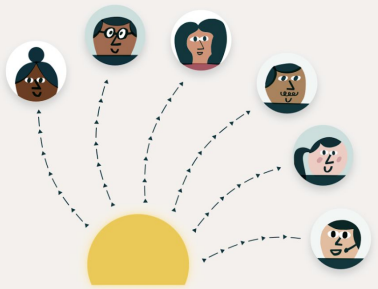
zendesk ❤ redis

sunshine conversations

built in Montréal

## Reach users wherever they are with a single API

Sunshine Conversations' universal messaging API lets you quickly and easily communicate with users on all the major chat platforms.



# what is redis

“Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker.”

(In the creator’s own words)

"Redis is an open source (BSD licensed), in-memory data structure store, used as a database, **cache** and message broker."

(In the creator's own words)

aws ElastiCache

Azure Cache for Redis

"Redis is an open source (BSD licensed), in-memory **data structure store**, used as a database, cache and message broker."

(In the creator's own words)

"Redis is an open source (BSD licensed), **in-memory** data structure store, used as a database, cache and message broker."

(In the creator's own words)

# in-memory does not mean ephemeral

redis also persists to disk

- full snapshot format (RDB)
- append only file format (AOF) “binlog”

# in-memory does not mean ephemeral

configure your durability needs

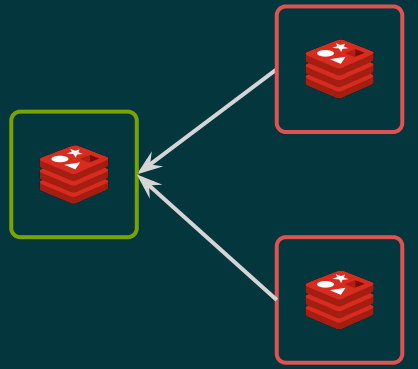
it's a tradeoff with throughput

# in-memory does not mean ephemeral

redis is also highly available (HA)

supports replication

supports sharding (aka clustering)



# the commands

# alphabet soup

**SET** key value [expiration EX seconds|PX milliseconds] [NX|XX]

Set the string value of a key

**BLPOP** key [key ...] timeout

Remove and get the first element in a list, or block until one is available

**BRPOP** key [key ...] timeout

Remove and get the last element in a list, or block until one is available

**BRPOPLPUSH** source destination timeout

Pop a value from a list, push it to another list and return it; or block until one is available

**BZPOPMIN** key [key ...] timeout

Remove and return the member with the lowest score from one or more sorted sets, or block until one is available

**BZPOPMAX** key [key ...] timeout

Remove and return the member with the highest score from one or more sorted sets, or block until one is available

# HMSET

## HMSET key field value [field value ...]

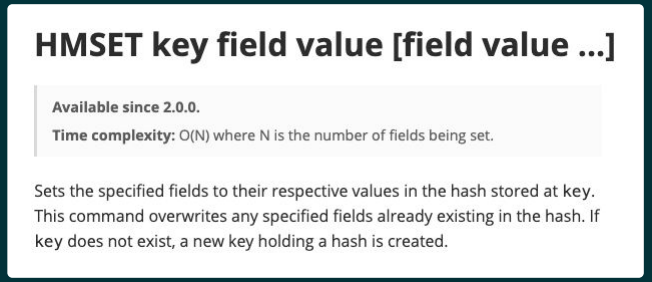
**Available since 2.0.0.**

**Time complexity:** O(N) where N is the number of fields being set.

Sets the specified fields to their respective values in the hash stored at key. This command overwrites any specified fields already existing in the hash. If key does not exist, a new key holding a hash is created.

# H M SET

H is for Hash, the type of data structure we're setting



**HMSET key field value [field value ...]**

**Available since 2.0.0.**

**Time complexity:** O(N) where N is the number of fields being set.

Sets the specified fields to their respective values in the hash stored at key. This command overwrites any specified fields already existing in the hash. If key does not exist, a new key holding a hash is created.

# H M SET

H is for Hash, the type of data structure we're setting

M is for multi, because we're setting multiple keys at once

## HMSET key field value [field value ...]

**Available since 2.0.0.**

**Time complexity:** O(N) where N is the number of fields being set.

Sets the specified fields to their respective values in the hash stored at key. This command overwrites any specified fields already existing in the hash. If key does not exist, a new key holding a hash is created.

# H M SET

H is for Hash, the type of data structure we're setting

M is for multi, because we're setting multiple keys at once

SET is for set… that's what we're doing

## HMSET key field value [field value ...]

**Available since 2.0.0.**

**Time complexity:** O(N) where N is the number of fields being set.

Sets the specified fields to their respective values in the hash stored at key. This command overwrites any specified fields already existing in the hash. If key does not exist, a new key holding a hash is created.

# H M SET

H is for Hash, the type of data structure we're setting

M is for multi, because we're setting multiple keys at once

SET is for set... that's what we're doing

## HMSET key field value [field value ...]

**Available since 2.0.0.**

**Time complexity:** O(N) where N is the number of fields being set.

Sets the specified fields to their respective values in the hash stored at key. This command overwrites any specified fields already existing in the hash. If key does not exist, a new key holding a hash is created.

```
> hmset myhash name andrew job developer
OK
```

# H M SET

H is for Hash, the type of data structure we're setting

M is for multi, because we're setting multiple keys at once

SET is for set... that's what we're doing

## HMSET key field value [field value ...]

**Available since 2.0.0.**

**Time complexity:** O(N) where N is the number of fields being set.

Sets the specified fields to their respective values in the hash stored at key. This command overwrites any specified fields already existing in the hash. If key does not exist, a new key holding a hash is created.

```
> hmset myhash name andrew job developer
OK
> hgetall myhash
1) "name"
2) "andrew"
3) "job"
4) "developer"
```

There are command prefixes for each data structure

```
key         = no prefix
hash        = H
list        = L or R (left or right)
sets        = S
sorted sets = Z
streams     = X
```

And a few other prefixes, postfixes and options

```
M  = get or set multiple
X  = postfix, only proceed if key exists
B  = blocking, if there's nothing to get
     wait indefinitely until there is
XX = only proceed if key exists
NX = only proceed if key does not exist
PX = set an expiration on this key
```

# alphabet soup

*BRPOPLPUSH ??* 👉

**`SET key value [expiration EX seconds|PX milliseconds] [NX|XX]`**
Set the string value of a key

**`BLPOP key [key ...] timeout`**
Remove and get the first element in a list, or block until one is available

**`BRPOP key [key ...] timeout`**
Remove and get the last element in a list, or block until one is available

**`BRPOPLPUSH source destination timeout`**
Pop a value from a list, push it to another list and return it; or block until one is available

**`BZPOPMIN key [key ...] timeout`**
Remove and return the member with the lowest score from one or more sorted sets, or block until one is available

**`BZPOPMAX key [key ...] timeout`**
Remove and return the member with the highest score from one or more sorted sets, or block until one is available

# BRPOPLPUSH

# B RPOP LPUSH

B is for blocking

# B RPOP LPUSH

B is for blocking

RPOP pop from the right side of one list

# B RPOP LPUSH

B is for blocking

RPOP pop from the right side of one list

LPUSH push the item to left side of another list

# B RPOP LPUSH

B is for blocking

RPOP pop from the right side of one list

LPUSH push the item to left side of another list

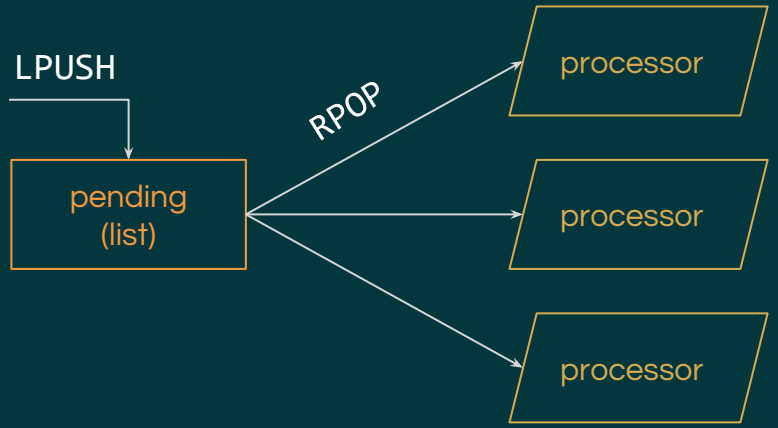
(do it atomically)

# B RPOP LPUSH

B is for blocking

RPOP pop from the right side of one list

LPUSH push the item to left side of another list

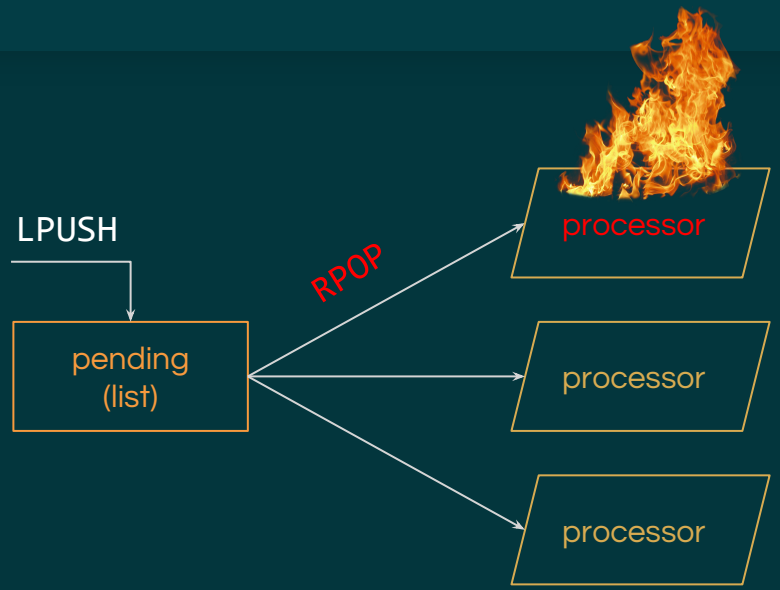


# B RPOP LPUSH

B is for blocking

RPOP pop from the right side of one list

LPUSH push the item to left side of another list

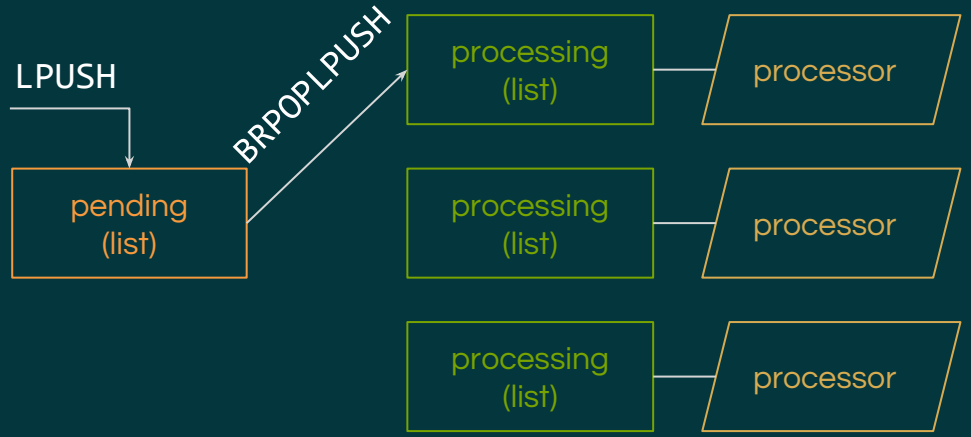


# B RPOP LPUSH

B is for blocking

RPOP pop from the right side of one list

LPUSH push the item to left side of another list

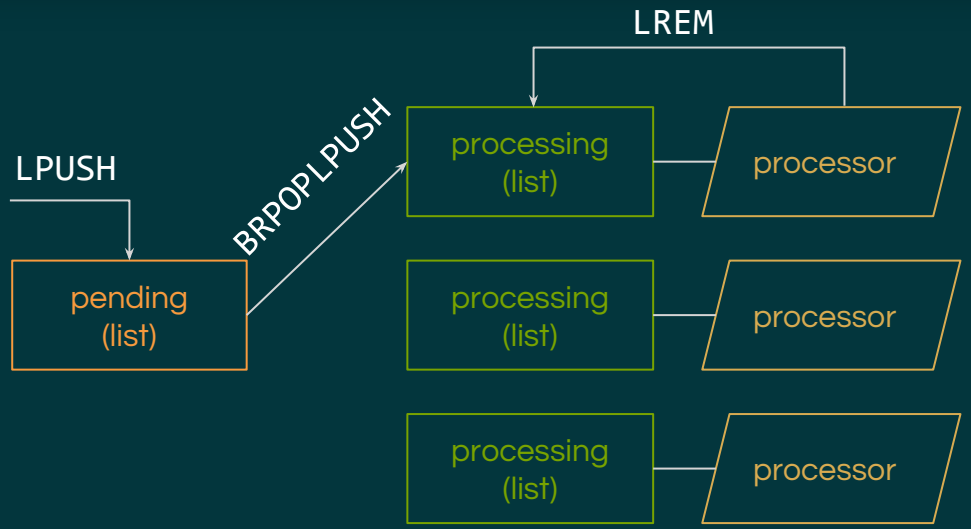


# B RPOP LPUSH

B is for blocking

RPOP pop from the right side of one list

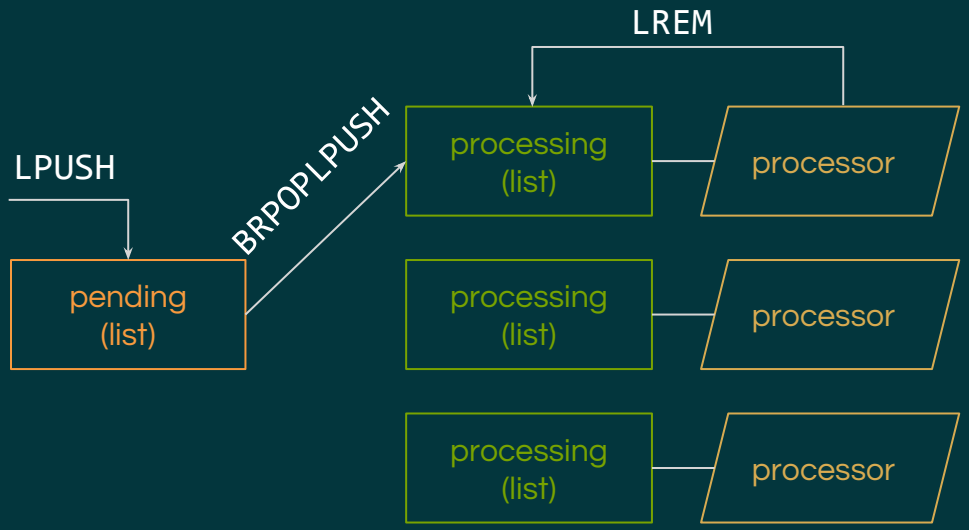
LPUSH push the item to left side of another list



## REDIS COMMANDS

```
127.0.0.1:6379> brpoplpush src dest 0
```

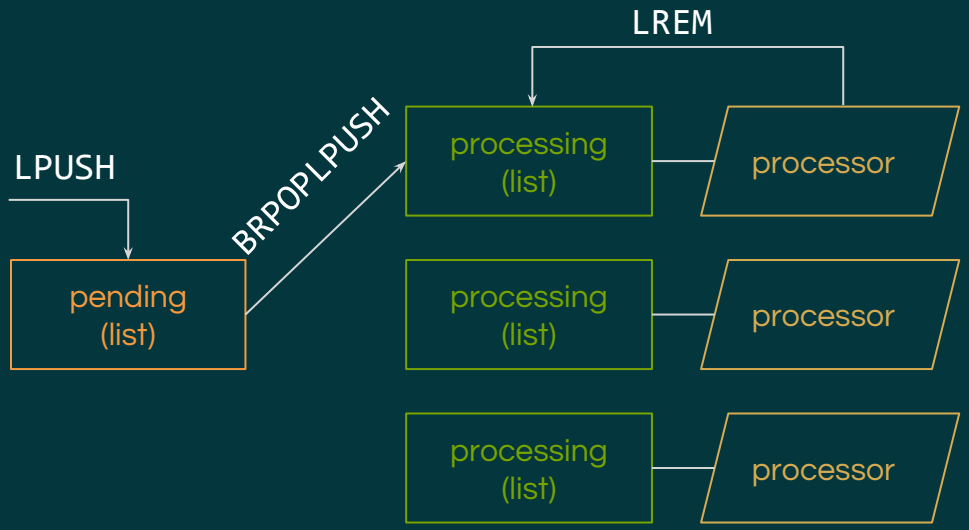
```
127.0.0.1:6379> rpush src he[value ...]
```



## REDIS COMMANDS

```
127.0.0.1:6379> brpoplpush src dest 0
"hello"
(4.90s)
127.0.0.1:6379>
```

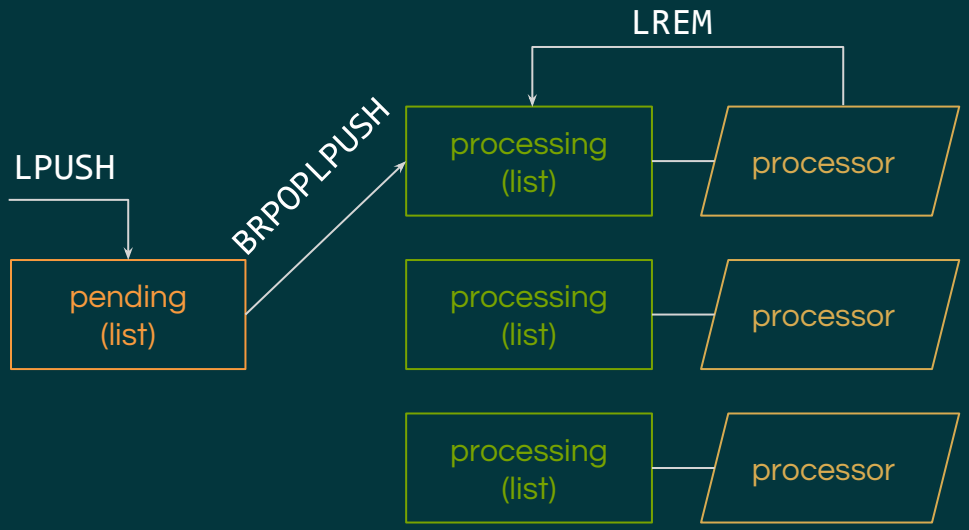
```
127.0.0.1:6379> rpush src hello
(integer) 1
127.0.0.1:6379>
```



## REDIS COMMANDS

```
127.0.0.1:6379> brpoplpush src dest 0
"hello"
(4.90s)
127.0.0.1:6379>
```

```
127.0.0.1:6379> rpush src hello
(integer) 1
127.0.0.1:6379> lrange dest 0 -1
1) "hello"
127.0.0.1:6379>
```

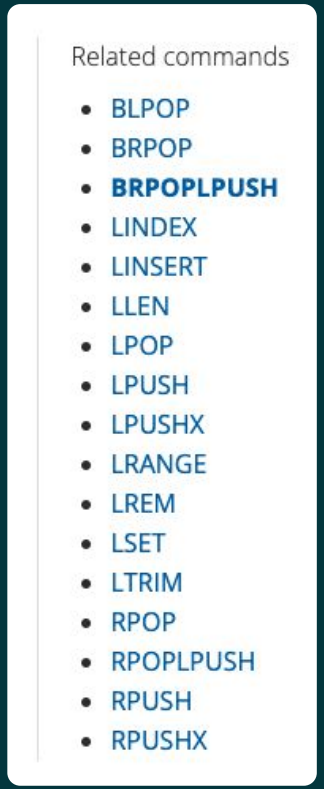


# alphabet soup

Related commands

- BLPOP
- BRPOP
- **BRPOPLPUSH**
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

# alphabet soup is starting to make sense



# the meat

# THE MEAT

1 ONE TIME TOKENS

2 LOCKS

3 RATE LIMITS

# 1. One time tokens

## user email verification

issue a new token:

```
> setex code:c17bd6 3600000 '{"act_id": "1234", "email": "user@example.org"}'
OK
```

issue a new token:

```
> setex code:c17bd6 3600000 '{"act_id": "1234", "email": "user@example.org"}'
OK
```

this is the key
the colon is just a delimiter to keep things organized

issue a new token:

```
> setex code:c17bd6 3600000 '{"act_id": "1234", "email": "user@example.org"}'
OK
```

the token itself

issue a new token:

```
> setex code:c17bd6 3600000 '{"act_id": "1234", "email": "user@example.org"}'
OK
```

the token itself
ms before it expires (1 hour)

issue a new token:

```
> setex code:c17bd6 3600000 '{"act_id": "1234", "email": "user@example.org"}'
OK
```

the token itself
ms before it expires (1 hour)
the value of the key

verify email is sent to the user
and the user clicks the link...

```
https://yoursite.com/verify_email?token=c17bd6
```

now we validate the token

```
> get code:c17bd6
"{\"act_id\": \"1234\", \"email\": \"user@example.org\"}"
```

and consume it

```
> del code:c17bd6
(integer) 1
```

we can do both **`get`** and **`del`** in one shot

you begin a redis transaction with **`multi`**
and you end it with **`exec`**

commands in between are
executed as an atomic transaction

```
> multi
OK

> any number of commands

> exec
array of command results
```

validate & consume the token

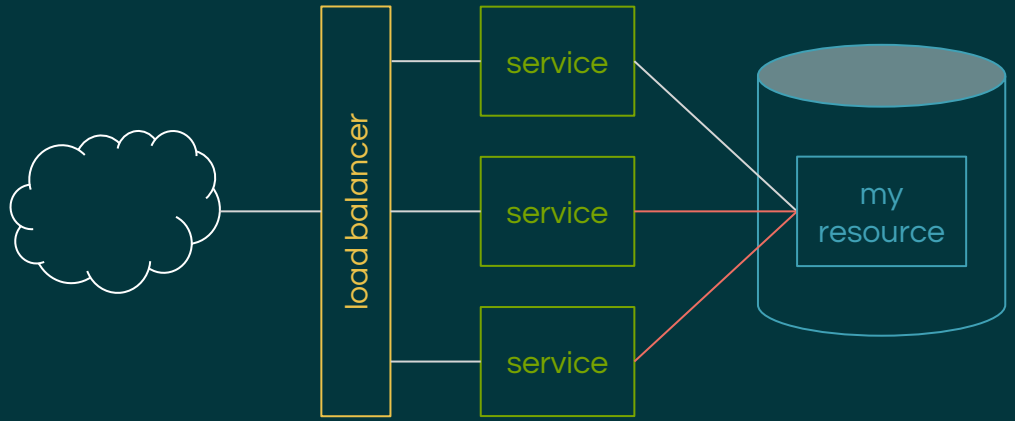
```
> multi
OK
> get code:c17bd6
QUEUED
> del code:c17bd6
QUEUED
> exec
1) "{\"act_id\": \"1234\", \"email\": \"user@example.org\"}"
2) (integer) 1
```

reject an invalid or consumed code

```
> multi
OK
> get code:invalid
QUEUED
> del code:invalid
QUEUED
> exec
1) (nil)
2) (integer) 0
```

# 2. Locks

why locks?



a redis key can be a lock

```
> set myresource ef1f63c NX PX 10000
OK
```

a redis key can be a lock

```
> set myresource ef1f63c NX PX 10000
OK
```

the resource id

a redis key can be a lock

```
> set myresource ef1f63c NX PX 10000
OK
```

the resource id
the key value

a redis key can be a lock

```
> set myresource ef1f63c NX PX 10000
OK
```

the resource id
the key value
NX = don't overwrite any existing key

a redis key can be a lock

```
> set myresource ef1f63c NX PX 10000
OK
```

the resource id
the key value
NX = don't overwrite any existing key
PX = expire this key in 10 seconds

`OK` means the lock was acquired

```
> set myresource ef1f63c NX PX 10000
OK
```

`(nil)` means the key already exists; try again later

```
> set myresource ef1f63c NX PX 10000
(nil)
```

`OK` means the lock was acquired

```
> set myresource ef1f63c NX PX 10000
OK
```

`(nil)` means the key already exists; try again later

```
> set myresource ef1f63c NX PX 10000
(nil)
```

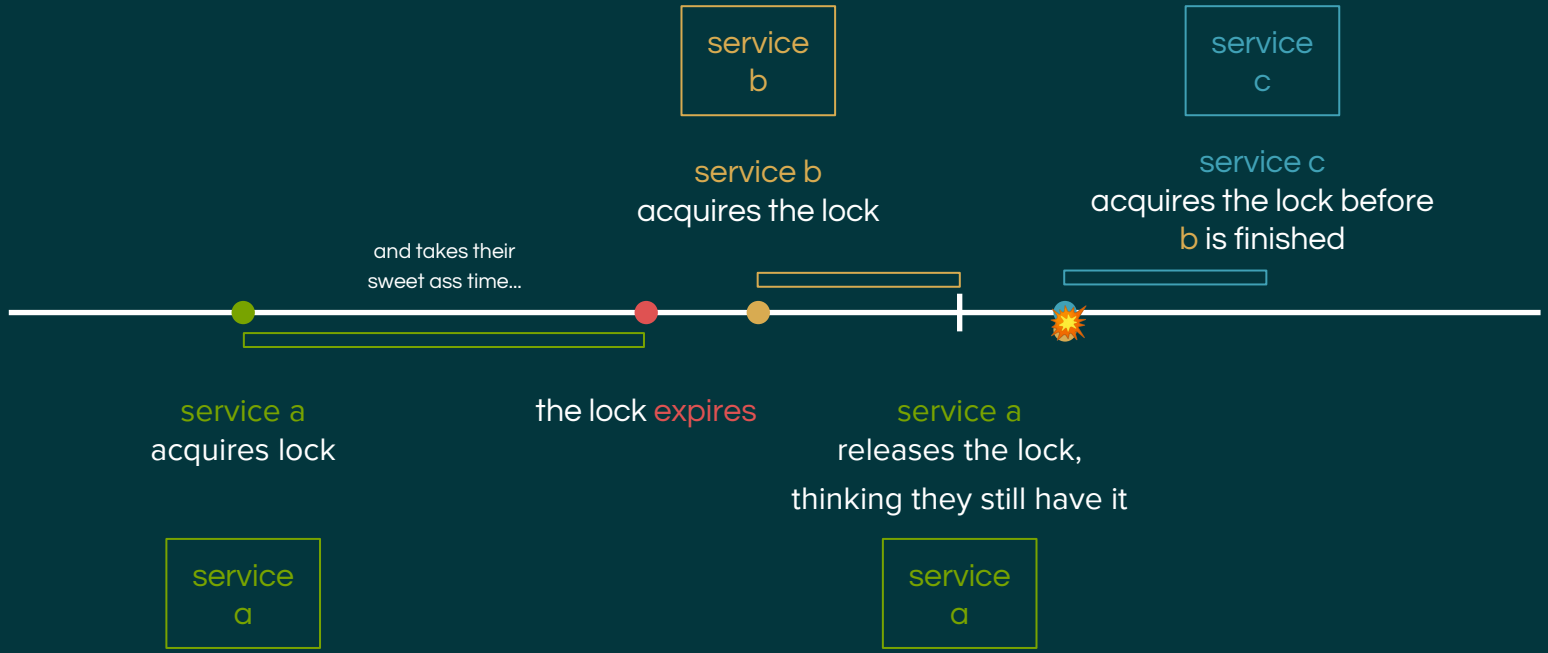
release the lock by deleting the key

```
> del myresource
(integer) 1
```

# Is it really that simple?

Unfortunately
not

# LOCKS


service b
service c
service b
acquires the lock
service c
acquires the lock before
b is finished
and takes their
sweet ass time...
service a
acquires lock
the lock expires
service a
releases the lock,
thinking they still have it
service a
service a

~~release the lock by deleting the key~~

```
~~> del myresource~~
~~(integer) 1~~
```

we need some way to check that we have the lock before releasing it

and we somehow need to do this part transactionally...

```
lock_value = crypto.random()

> set myresource lock_value NX PX 10000
OK

do_my_stuff()

> get myresource
"ef1f63c"

if (lock_value === “ef1f63c”) {
  > del myresource
}
```

# Lua scripting!

**EVAL** and **EVALSHA** commands allow you to run some logic remotely in redis

Redis will execute your lua script atomically

our safe release lua script looks like this:

```
if redis.call('get', KEYS[1]) == ARGV[1]
then return redis.call('del', KEYS[1])
else return 0 end
```

our safe release lua script looks like this:

```
if redis.call('get', KEYS[1]) == ARGV[1]
then return redis.call('del', KEYS[1])
else return 0 end
```

lua scripts use `redis.call()` to invoke redis commands

our safe release lua script looks like this:

```
if redis.call('get', KEYS[1]) == ARGV[1]
then return redis.call('del', KEYS[1])
else return 0 end
```

arguments are passed to the script as key value pairs

we pass the resource key as `KEYS[1]`
and the lock value `ARGV[1]`

our safe release lua script looks like this:

```
if redis.call('get', KEYS[1]) == ARGV[1]
then return redis.call('del', KEYS[1])
else return 0 end
```

arguments are passed to the script as key value pairs

we pass the resource key as `KEYS[1]`
and the lock value `ARGV[1]`

*(and yes... Lua has 1-based indexes 🙃)*

our safe release lua script looks like this:

```
if redis.call('get', KEYS[1]) == ARGV[1]
then return redis.call('del', KEYS[1])
else return 0 end
```

acquire the lock (after generating a lock value)

```
> set myresource ef1f63c NX PX 10000
OK
```

release the lock (safely)

```
> eval "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del',
KEYS[1]) else return 0 end" 1 myresource , ef1f63c
(integer) 1
```

acquire the lock (after generating a lock value)

```
> set myresource ef1f63c NX PX 10000
OK
```

release the lock (safely)  the number of keys I'm passing

```
> eval "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del',
KEYS[1]) else return 0 end" 1 myresource , ef1f63c
(integer) 1
```

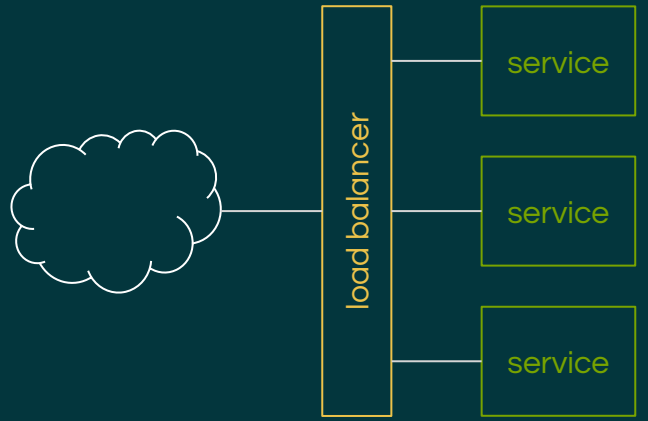
with **SCRIPT LOAD** and **EVALSHA** you don't have send your entire script payload every time

```
> script load "if redis.call('get', KEYS[1]) == ARGV[1] then return
redis.call('del', KEYS[1]) else return 0 end"
"e9f69f2beb755be68b5e456ee2ce9aadfbc4ebf4"

> evalsha e9f69f2beb755be68b5e456ee2ce9aadfbc4ebf4 1 myresource , ef1f63c
(integer) 1
```

# 3. Rate limits

rate limits protect your service from traffic spikes

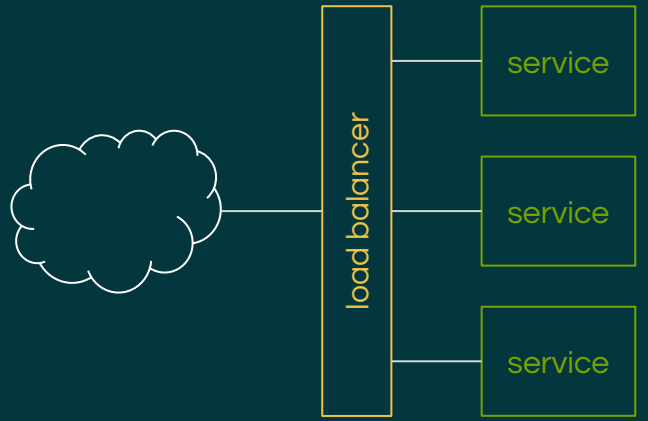
you'd typically them in your load balancer or web server,
e.g. ELB or NGINX



but perhaps your rate limit criteria is complex

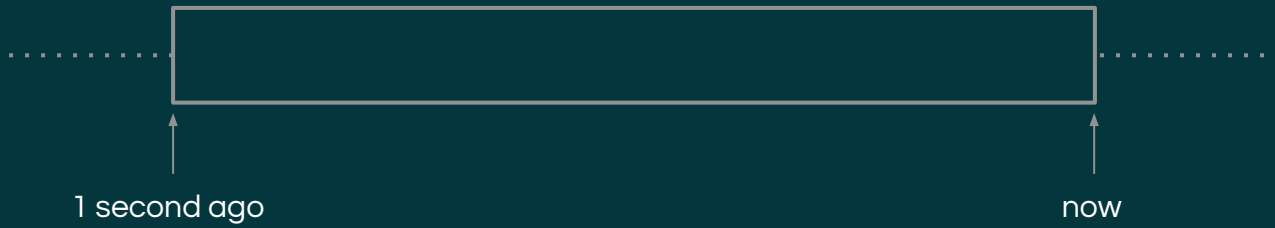
e.g. what about a GraphQL **query complexity** rate limit

redis can help

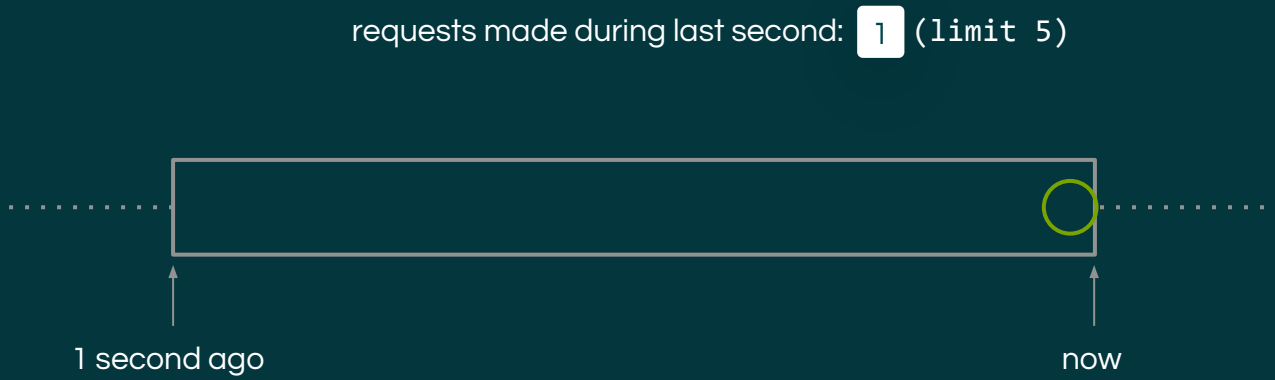


let's impose a limit of **5 requests per second**
starting simple, consider a **rolling window model**

requests made during last second: `0` `(limit 5)`

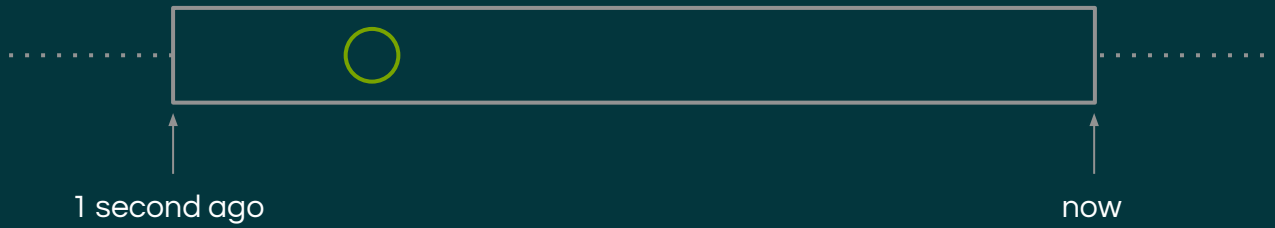


new requests come in at the front of the window



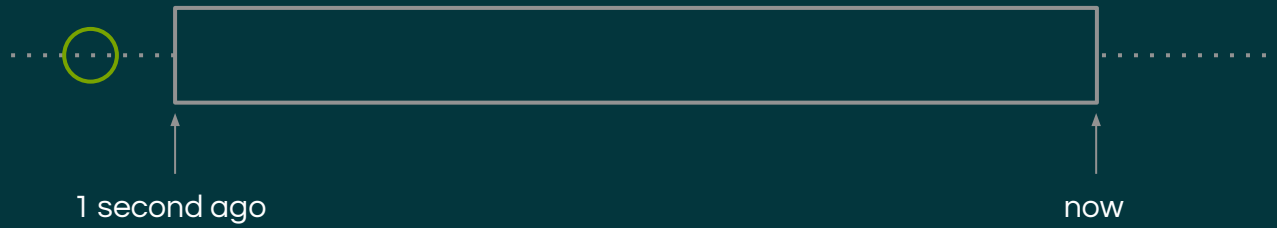
and as time goes on they drift to the back

requests made during last second: `1` (`limit 5`)



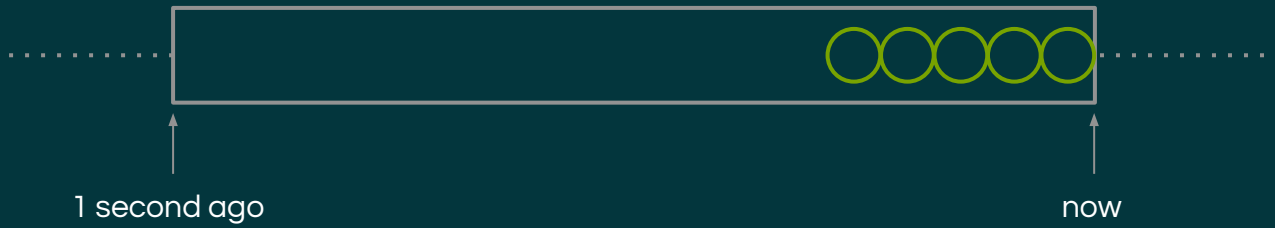
and out of the window

requests made during last second: `0` (`limit 5`)

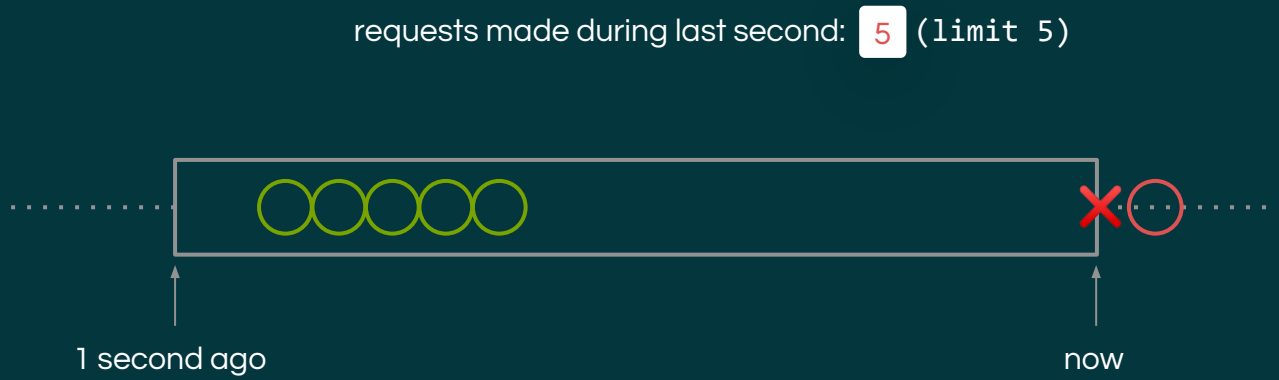


we can accept a burst of many requests at once

requests made during last second: 5 (limit 5)

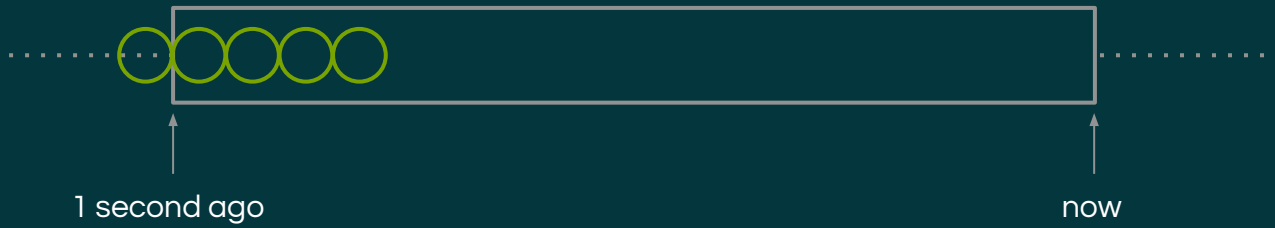


but some time needs to pass before we accept any more



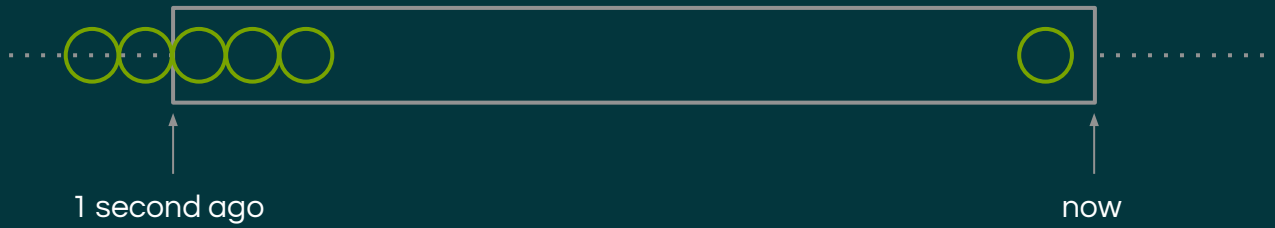
but some time needs to pass before we accept any more

requests made during last second: `4` (`limit 5`)



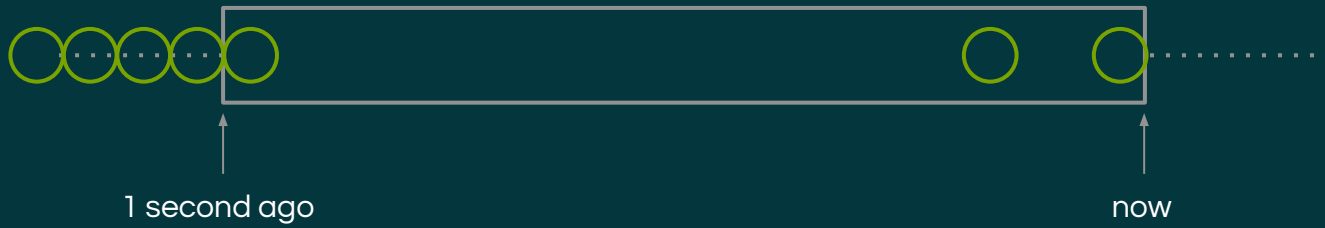
but some time needs to pass before we accept any more

requests made during last second: `4` (`limit 5`)



but some time needs to pass before we accept any more

requests made during last second: `3` `(limit 5)`



# Let’s make a rolling window with redis

we’ll use lua

# Let’s make a rolling window with redis

we’ll use lua
and sorted sets

# Sorted Sets 101

## Adding members (with scores)

```
> zadd leaderboard 1000000 parzival
(integer) 1
> zadd leaderboard 900000 art3mis
(integer) 1
```

zrevrange: list members from start (0) to finish (-1), in reverse order

```
> zrevrange leaderboard 0 -1 withscores
 1) "parzival"
 2) "1000000"
 3) "art3mis"
 4) "900000"
 5) "ioi987427"
 6) "0"
 7) "ioi677427"
 8) "0"
 9) "ioi423417"
10) "0"
```

## Removing individual members

```
> zrem leaderboard ioi987427
(integer) 1
> zrevrange leaderboard 0 -1 withscores
1) "parzival"
2) "1000000"
3) "art3mis"
4) "900000"
5) "ioi677427"
6) "0"
7) "ioi423417"
8) "0"
```

Removing a range of members
remove all members whose score is between -inf and 0, inclusive:

```
> zremrangebyscore leaderboard -inf 0
(integer) 2
```

```
> zrevrange leaderboard 0 -1 withscores
1) "parzival"
2) "1000000"
3) "art3mis"
4) "900000"
```

Tell me how many members there are (cardinality)

```
> zcard leaderboard
(integer) 2
```

Now that you know `ZADD`, `ZCARD` and `ZREMRANGEBYSCORE`, let's look at a Lua script

```
local rateLimit = 5

local rateKey = KEYS[1]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    return 1;
end
```

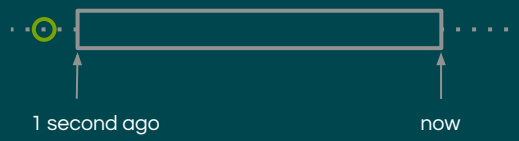
👆Much inspiration was had from Sahil Jadon's article *Rate Limiting using Redis Lists and Sorted Sets*
https://medium.com/@sahiljadon/rate-limiting-using-redis-lists-and-sorted-sets-9b42bc192222

```lua
local rateLimit = 5

local rateKey = KEYS[1]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    return 1;
end
```

Set the rate limit, receive args
Timestamp (ts) is passed in as an arg

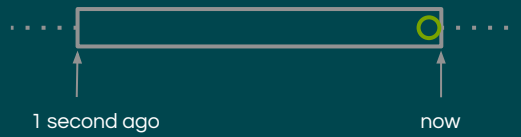
```
local rateLimit = 5

local rateKey = KEYS[1]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    return 1;
end
```

remove any members that are older than our rolling window



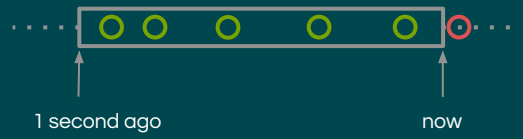
```
local rateLimit = 5

local rateKey = KEYS[1]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    return 1;
end
```

if there's room in the window, accept the request



```lua
local rateLimit = 5

local rateKey = KEYS[1]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    return 1;
end
```

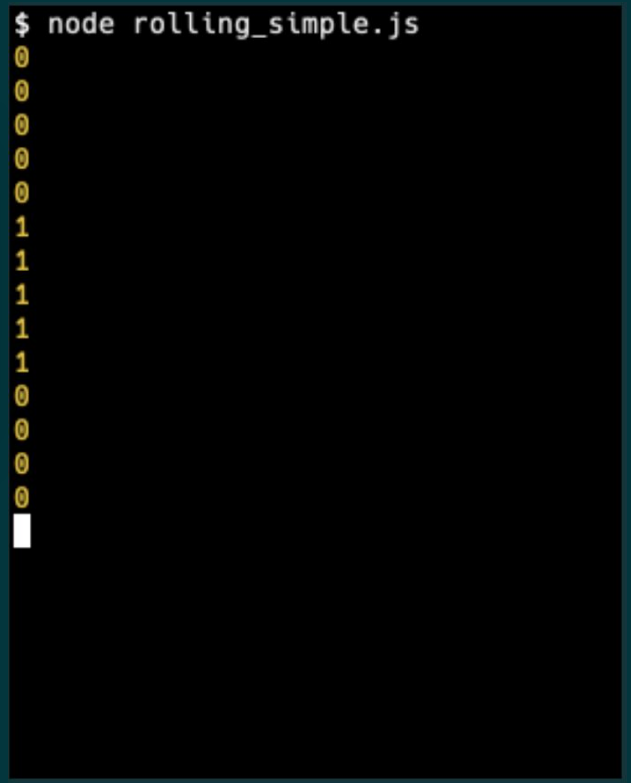
otherwise, reject the request and return **1** to indicate



# Let’s test it

`eval` the script 10 times per second
with a rate limit of 5, half get accepted

```
const script = fs.readFileSync('rolling_simple.lua', 'utf8');
const rc = redis.createClient();

setInterval(() => {
    rc.eval(script, 1, 'myresource', Date.now()).then(console.log);
}, 100);
```



# What about retry advice?

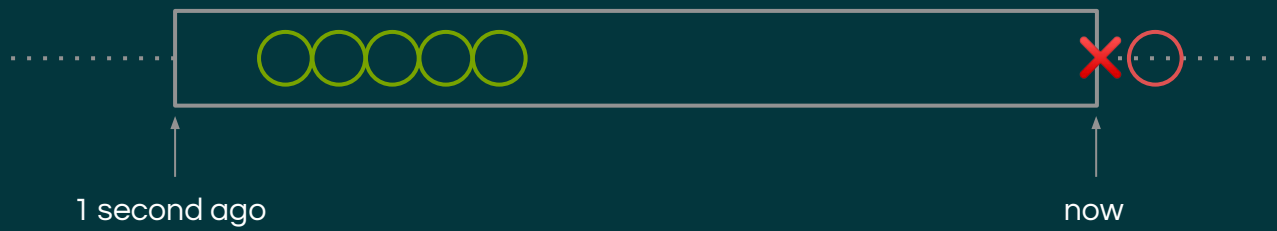
when returning a 429 HTTP rate limit error it's courteous to include a `Retry-After` header telling the client how long to wait

```
HTTP/1.1 429 Too Many Requests
Content-Type: text/html
Retry-After: 3600
```

# Let’s compute a simple `Retry-After` with redis

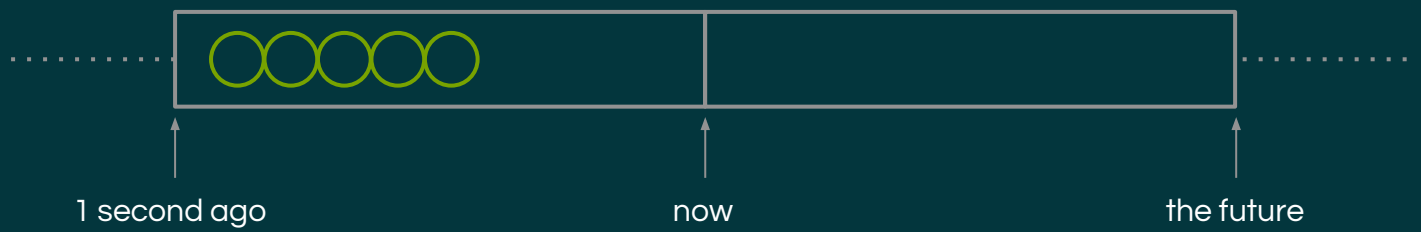
remember our rolling window?

requests made during last second: 5 (`limit 5`)



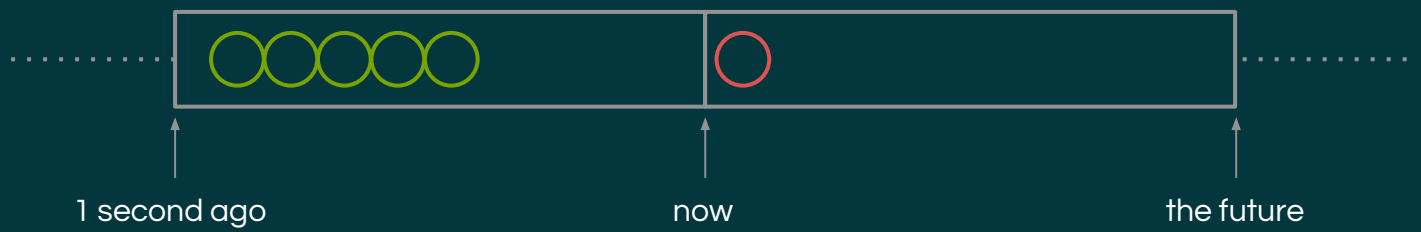
remember our rolling window?
let's add a *future* window

requests made during last second: 5 (`limit 5`)



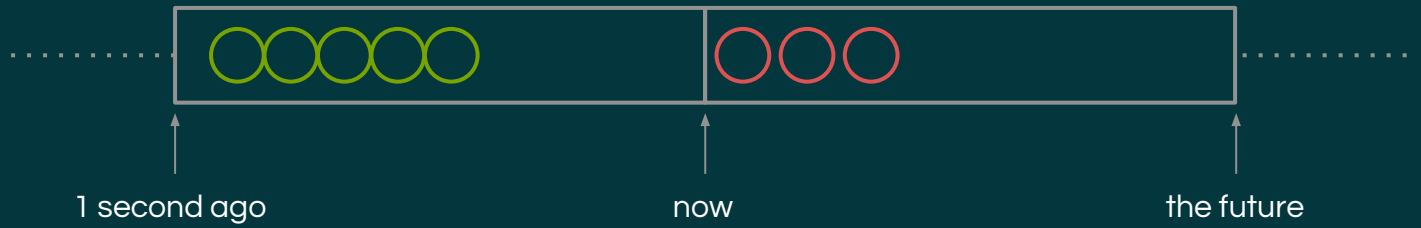
when request rates exceed our limit, add to our future window

requests made during last second: 5 (limit 5)

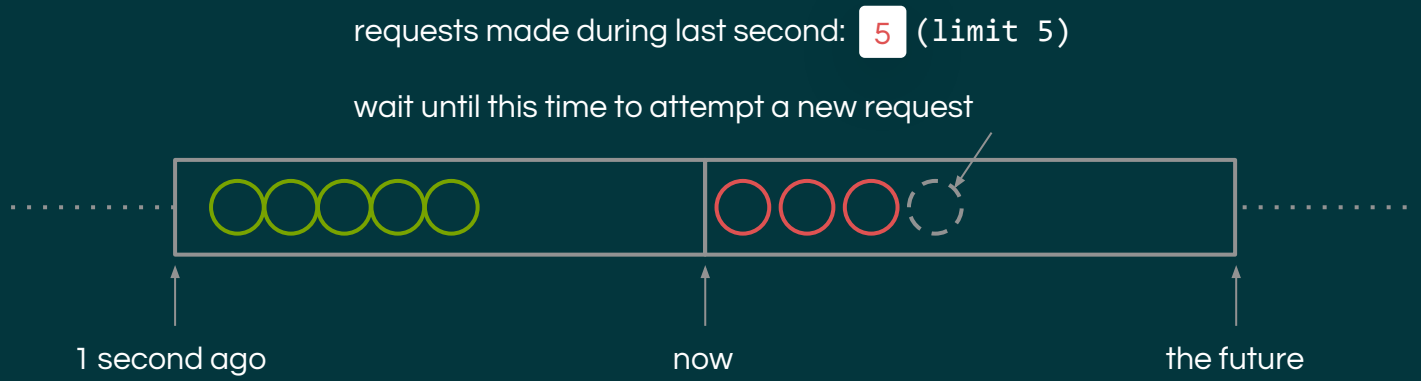


when request rates exceed our limit, add to our future window

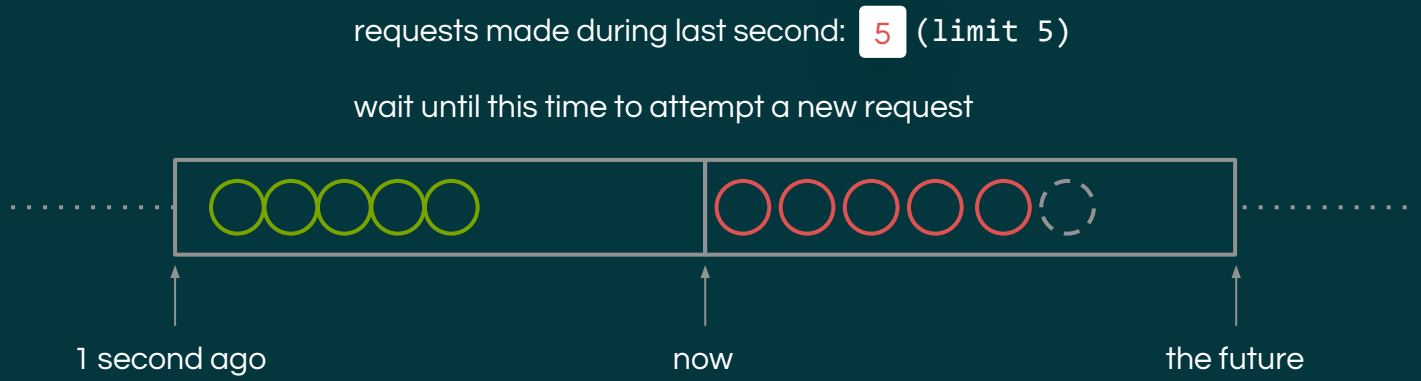
requests made during last second: 5 (limit 5)



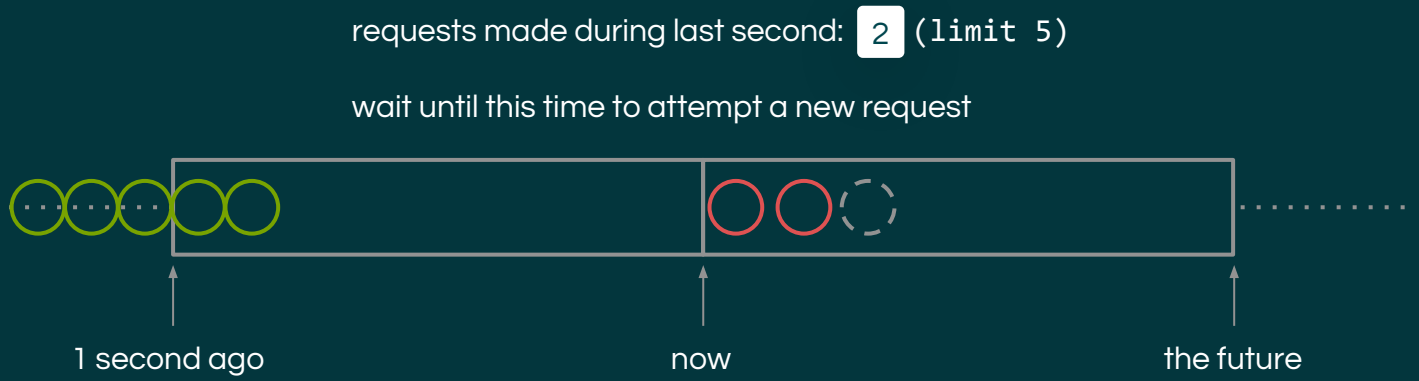
the last value in our future window is the retry advice



as the rate limit continues to be exceeded the Retry-After will continue to grow



and as time goes on our window moves leftward



# RATE LIMITS

```lua
local rateLimit = 5

local rateKey = KEYS[1]
local retryKey = KEYS[2]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)
redis.call('ZREMRANGEBYSCORE', retryKey, '-inf', ts)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    local interval = 1000 / rateLimit

    local next = ts + interval;
    local last = redis.call('ZRANGE', retryKey, -1, -1, 'WITHSCORES')

    if type(last) == 'table' and #last > 0 then
        next = tonumber(last[#last]) + interval
    end

    redis.call('ZADD', retryKey, next, next);

    return next - ts;
end
```

zendesk

```lua
local rateLimit = 5

local rateKey = KEYS[1]
local retryKey = KEYS[2]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)
redis.call('ZREMRANGEBYSCORE', retryKey, '-inf', ts)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    local interval = 1000 / rateLimit

    local next = ts + interval;
    local last = redis.call('ZRANGE', retryKey, -1, -1, 'WITHSCORES')

    if type(last) == 'table' and #last > 0 then
        next = tonumber(last[#last]) + interval
    end

    redis.call('ZADD', retryKey, next, next);

    return next - ts;
end
```

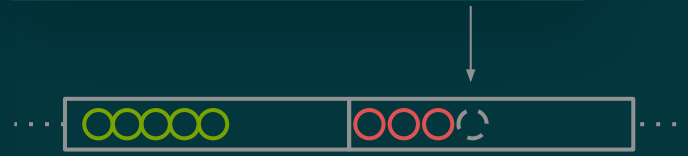
this should all look familiar, except we have two sorted sets now, `rateKey` and `retryKey`

# RATE LIMITS

```lua
local rateLimit = 5

local rateKey = KEYS[1]
local retryKey = KEYS[2]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)
redis.call('ZREMRANGEBYSCORE', retryKey, '-inf', ts)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    local interval = 1000 / rateLimit

    local next = ts + interval;
    local last = redis.call('ZRANGE', retryKey, -1, -1, 'WITHSCORES')

    if type(last) == 'table' and #last > 0 then
        next = tonumber(last[#last]) + interval
    end

    redis.call('ZADD', retryKey, next, next);

    return next - ts;
end
```

rate limit reached!

find the last member of the retry sorted set and add our interval to it

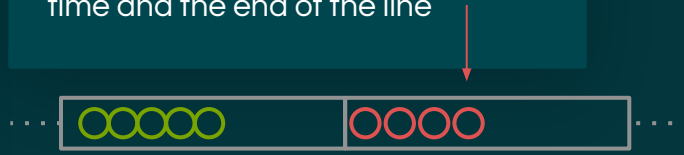


zendesk

# RATE LIMITS

```
local rateLimit = 5

local rateKey = KEYS[1]
local retryKey = KEYS[2]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)
redis.call('ZREMRANGEBYSCORE', retryKey, '-inf', ts)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    local interval = 1000 / rateLimit

    local next = ts + interval;
    local last = redis.call('ZRANGE', retryKey, -1, -1, 'WITHSCORES')

    if type(last) == 'table' and #last > 0 then
        next = tonumber(last[#last]) + interval
    end

    redis.call('ZADD', retryKey, next, next);

    return next - ts;
end
```

add the new member and return the difference between the current time and the end of the line



zendesk

# Let’s test it

node.js script:

- runs the lua script every 100ms
- if rate limited, schedule a retry based on the returned delay
- periodically check # of requests accepted per second

```
  rate limited, retry after: 6077
  rate limited, retry after: 6196
  rate limited, retry after: 6376
  rate limited, retry after: 6472
requests accepted in the last last 1000ms: 5
  rate limited, retry after: 6595
  rate limited, retry after: 6770
  rate limited, retry after: 6869
  accepted, waited: 3395
  rate limited, retry after: 6968
  accepted, waited: 0
  accepted, waited: 3536
  rate limited, retry after: 6965
  accepted, waited: 0
  rate limited, retry after: 6994
  accepted, waited: 0
  rate limited, retry after: 7053
  rate limited, retry after: 7196
  rate limited, retry after: 7353
  rate limited, retry after: 7453
^C
```

# Caveats?

```
local rateLimit = 5

local rateKey = KEYS[1]
local retryKey = KEYS[2]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)
redis.call('ZREMRANGEBYSCORE', retryKey, '-inf', ts)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    local interval = 1000 / rateLimit

    local next = ts + interval;
    local last = redis.call('ZRANGE', retryKey, -1, -1, 'WITHSCORES')

    if type(last) == 'table' and #last > 0 then
        next = tonumber(last[#last]) + interval
    end

    redis.call('ZADD', retryKey, next, next);

    return next - ts;
end
```

# Caveats?

no limit
cap the retry sorted set

```
local rateLimit = 5

local rateKey = KEYS[1]
local retryKey = KEYS[2]
local ts = tonumber(ARGV[1])

redis.call('ZREMRANGEBYSCORE', rateKey, '-inf', ts - 1000)
redis.call('ZREMRANGEBYSCORE', retryKey, '-inf', ts)

if redis.call('ZCARD', rateKey) < rateLimit then
    redis.call('ZADD', rateKey, ts, ts)
    return 0;
else
    local interval = 1000 / rateLimit

    local next = ts + interval;
    local last = redis.call('ZRANGE', retryKey, -1, -1, 'WITHSCORES')

    if type(last) == 'table' and #last > 0 then
        next = tonumber(last[#last]) + interval
    end

    redis.call('ZADD', retryKey, next, next);

    return next - ts;
end
```

# Redis:
# not just a cache


# Thank you

alavers
alavers
alavers@zendesk.com

# bonus

# Redis wire protocol

RESP (REdis Serialization Protocol)

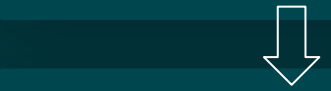
TCP 6379
encoded parts `strings, integers, arrays, errors`
parts terminated with CRLF `/r/n`

part prefixes:

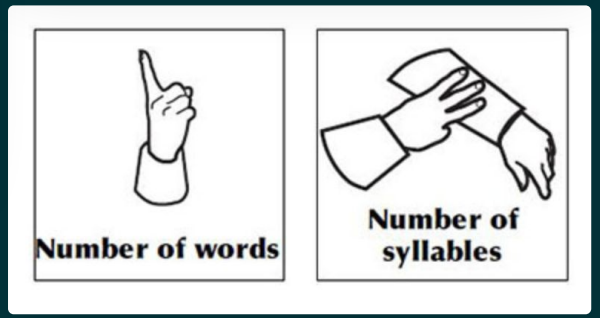
```
prefixes:
----------------------
+  = simple string
-  = error
:  = integer
$n = string of n bytes
*n = array of n items
```

```
examples:
----------------------
+OK\r\n
-WRONGTYPE...\r\n
:1\r\n
$3\r\nget\r\n
*2\r\n:1\r\n:1\r\n
```

```
> get foo
bar
```



```
*2\r\n$3\r\nget\r\n$3\r\nfoo
$3\r\nbar
```




# Thank you

alavers
alavers
alavers@zendesk.com