High Volume Ratelimiting with Redis

by

Triton Ho

Content

- Our problem
- v0 approach
- v1 approach

Our problem

- Unpredictable events cause sudden burst of QPS
- Auto-scaling is not fast enough
- Database is painful to scale
  - Especially during peak hours
  - Especially with m***odb
- Solution: HTTP 429, Too Many Requests

Why not Nginx

- Users play 30mins game in our app
  - Interruption causes horribly bad UX
- When QPS over server capacity
  - Allow requests from existing users
  - Block requests from new users
- new users = no request in previous 30mins
- Nginx doesn't support such rate-limiting

Technical Requirements

- When total active user > X, system enters "throttling mode"
- During throttling mode
  - Check for AccessToken
  - If the AccessToken is used in previous 30mins

    ```
        Allows it
    else
        HTTPS 429
    ```

V0 approach in Redis

- hs_20190309:2300
  - Init TTL: 60min
  - KeyValues
    - TokenA: 20190309:231012
    - TokenB: 20190309:231839
    - TokenC: 20190309:232912
- hs_20190309:2330
  - Init TTL: 60min
  - KeyValues
    - TokenB: 20190309:233139
    - TokenC: 20190309:233312
    - TokenD: 20190309:233454

v0 approach

- Simple, but not working
- Maintain multiple hashset in redis
  - Key = AccessToken, Value = lastUsedTs
- Updates lastUsedTs in every request
- Active User ~= math.Max(HLEN(hs_current), HLEN(hs_previous))
- During system throttling, check for “current” and “prevous” hashset

Sound bad, and epic fails in v0

- Rate-limiting’s goal
  - protecting the redis(and other components) from QPS burst
- But in v0 design
  - ALL requests need one redis call before rejected
- All requests need update on single hashSet
  - Super hotspot in redis cluster

Goal in v1

- The redis call MUST be independent of QPS
- Use local memory in application server instead of redis
- Constant local memory usage

- Seems mission-impossible......

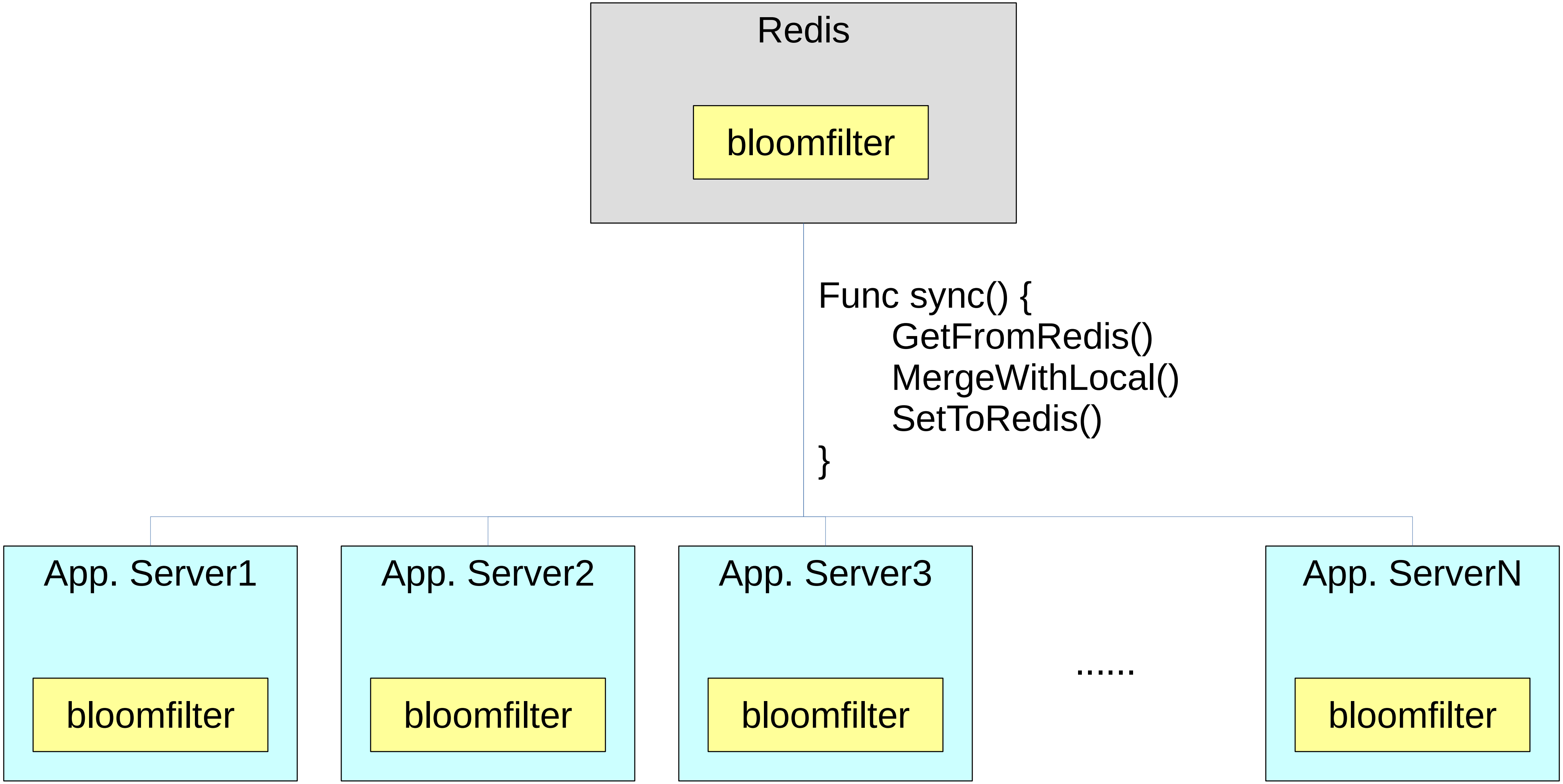
Bloomfilter data structure

- Constant memory usage
- Add(string s)
  - Add string s into the bloomfilter
- Test(string s) bool
  - Test if string s exists
  - may return false positive
- AppxCount() int
  - Return the approximation of the number of stored items
- Merge(Bloomfilter bf)
  - Merge two bloomfilter

Why bloomfilter helpful?

- Little false positive doesn't matter
- AppxCount() gives fast approximation of active users in the system
- Merge use bitwise OR operation on byte array
  - Allow super fast synchronization between application server
  - Gzipped byte array is small, small network bandwidth requirement

V1 (simplified) architecture



Redis

bloomfilter

```
Func sync() {
        GetFromRedis()
        MergeWithLocal()
        SetToRedis()
}
```

App. Server1

bloomfilter

App. Server2

bloomfilter

App. Server3

bloomfilter

......

App. ServerN

bloomfilter

App. server Synchronization

- Use redis as central storage
  - No extra backend component
    a.k.a. no extra approval from manager, smile~
  - Every second, the app. server sync the local copy with redis
- Opimistic lock is used to protect concurrent sync from multiple app. server
  - HashSet is used instead of KeyValue in redis
  - “ts” field for timpstamp,
  - “value” field for bloomfilter content
  - Evalsha() is needed to run the lua script in redis

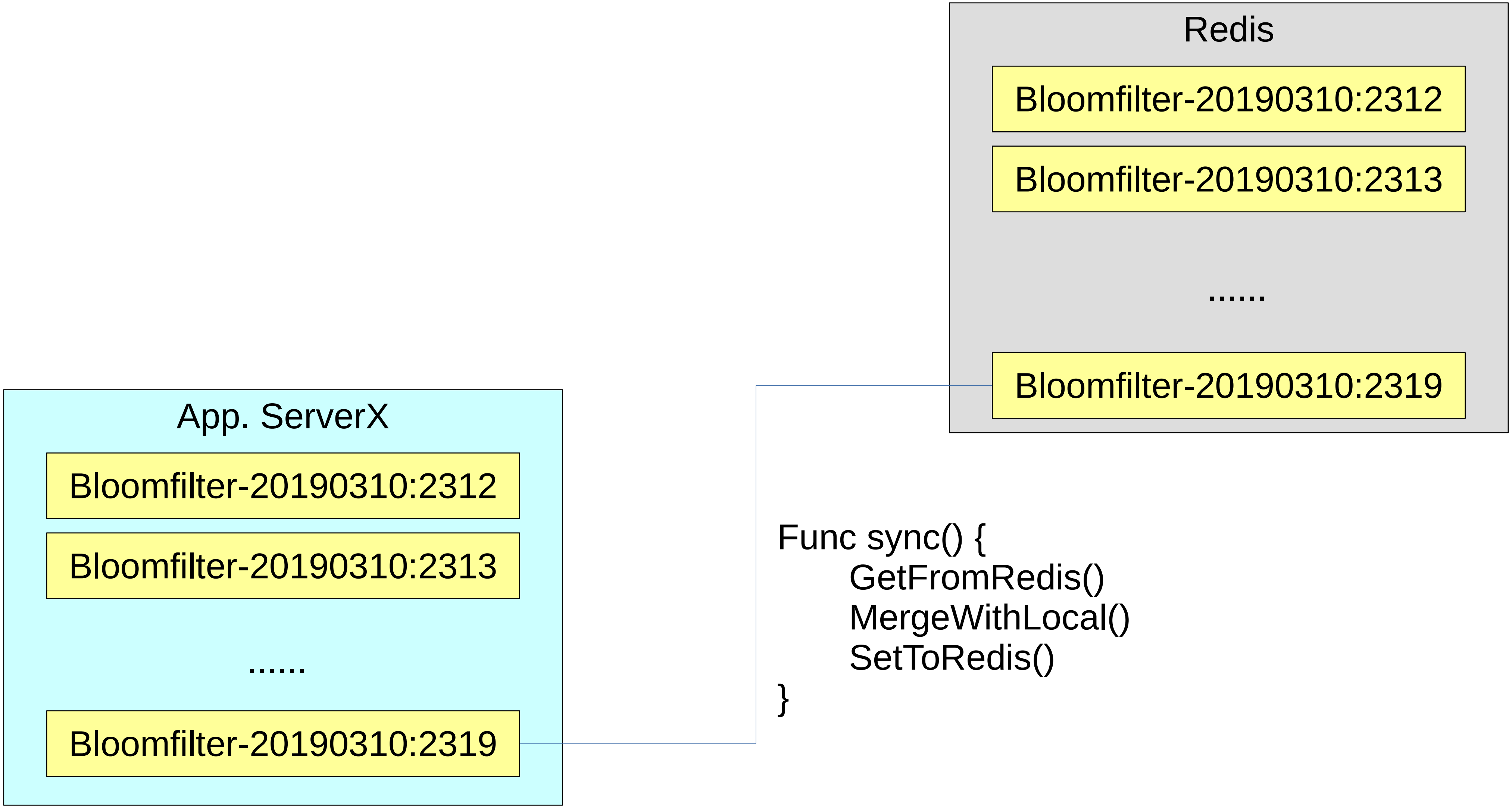
Concurrency in app. server

- RW-lock is used
  - Allow concurrency read, but single-thread update of bloomfilter
- To minimize latency, the update of bloomfilter is deferred
  - Buffered channel in Golang is great~

Removing old AccessToken

- Bloomfilter has no “expiration” mechanism
- “log rotation” on bloomfilter
  - Keep 30 bloomfilter
  - in every minute, remove the oldest one and create a new one
- New AccessToken add to the latest bloomfilter
  - Thus, only latest bloomfilter need to keep sync. with redis

V1 architecture



Redis

Bloomfilter-20190310:2312

Bloomfilter-20190310:2313

......

Bloomfilter-20190310:2319

App. ServerX

Bloomfilter-20190310:2312

Bloomfilter-20190310:2313

......

Bloomfilter-20190310:2319

```
Func sync() {
        GetFromRedis()
        MergeWithLocal()
        SetToRedis()
}
```

fails in v1

- Optimistic lock description in wikipedia
  - OCC is generally used in environments with low data contention
  - if contention for data resources is frequent, the cost of repeatedly restarting transactions hurts performance significantly
- The sync. conflict increase exponential with app. server number.
  - Caused abnormal high bandwidth usage during peak hours

v1.1 modification

- During optimistic lock conflict, the app. server simply give up the SetToRedis()
  - The sync. is done in every second, some failed SetToRedis() can be tolerated

Lessons learnt

- Performing redis call per request for ratelimiting is horribly bad idea
  - Your redis will fail during DDoS
  - Similarly, random-string AccessToken is BAD
- Be careful for hotspot in redis cluster
- Optimistic lock is suitable if the conflict rate is very low
- Algorithm and data structure has practical use
  - DO NOT SLEEP IN UNIVERSITY COURSES

More thought

- In the world of >10K QPS, consistency is usually sacrificed for performance
  - Bloomfilter is an appx algorithm
- Efficient use of app. server local memory is the key for high performance

Extras

- Can you use redis to implement a priority task queue?
  - Sorted Set should be avoided at all cost
  - For fault tolerance appliation, you should use rabbitMQ instead

End~