Efficient Redis Expiration Strategy For Session Storage

Guochao Xie

The Chinese University of Hong Kong, Shenzhen

2020-03-18

Introduction

1. *Session* is widely used in web applications to store users' **transitory** information.
2. *Redis* is the **most common** approach to store the session data in a cluster.
3. However, the current **expiration strategy** adopted in *Redis* still has some space to improve.

Background

Session Storage in Web Applications

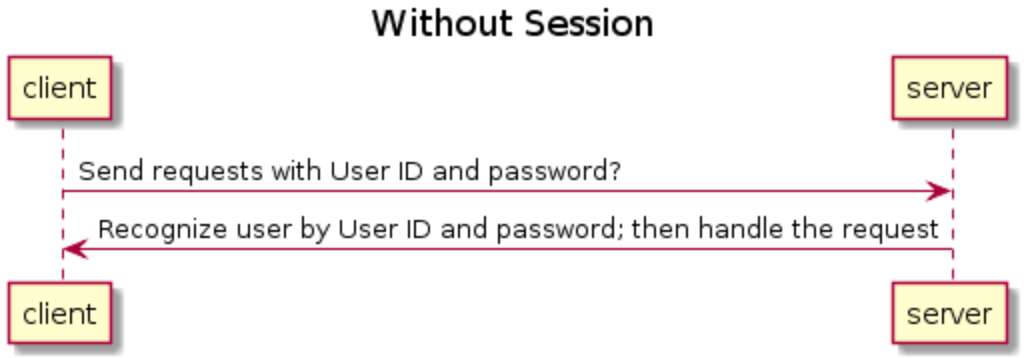
Session is a **temporary** and **interactive** information interchange between two or more communicating devices, or between a computer and user (Wikipedia).

Why Session is essential? Stateful Application vs Stateless Protocal

Modern web applications adopt *HTTP* as a standard protocol; however, *HTTP* is **stateless**.

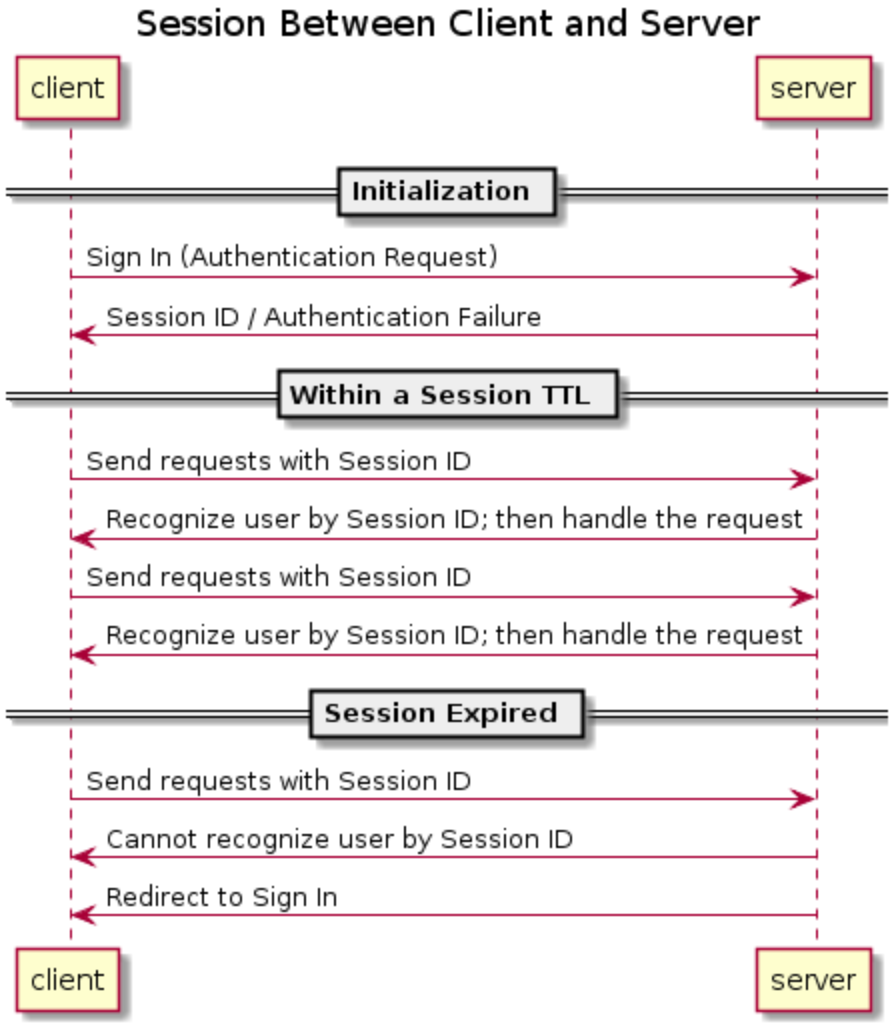
$\Rightarrow$ We need other methods to keep the information of **state**

$\Rightarrow$ ***Session***



If we do not use server-end session storage, we will have a lot of issues:

1. Security issue: Sending ID and password for every query increases the protential of steal.
2. Efficiency issue: Server has to do authentication for every query $\Rightarrow$ Slow!


Session Between Client and Server
client
server
Initialization
Sign In (Authentication Request)
Session ID / Authentication Failure
Within a Session TTL
Send requests with Session ID
Recognize user by Session ID; then handle the request
Send requests with Session ID
Recognize user by Session ID; then handle the request
Session Expired
Send requests with Session ID
Cannot recognize user by Session ID
Redirect to Sign In
client
server

Benefits:

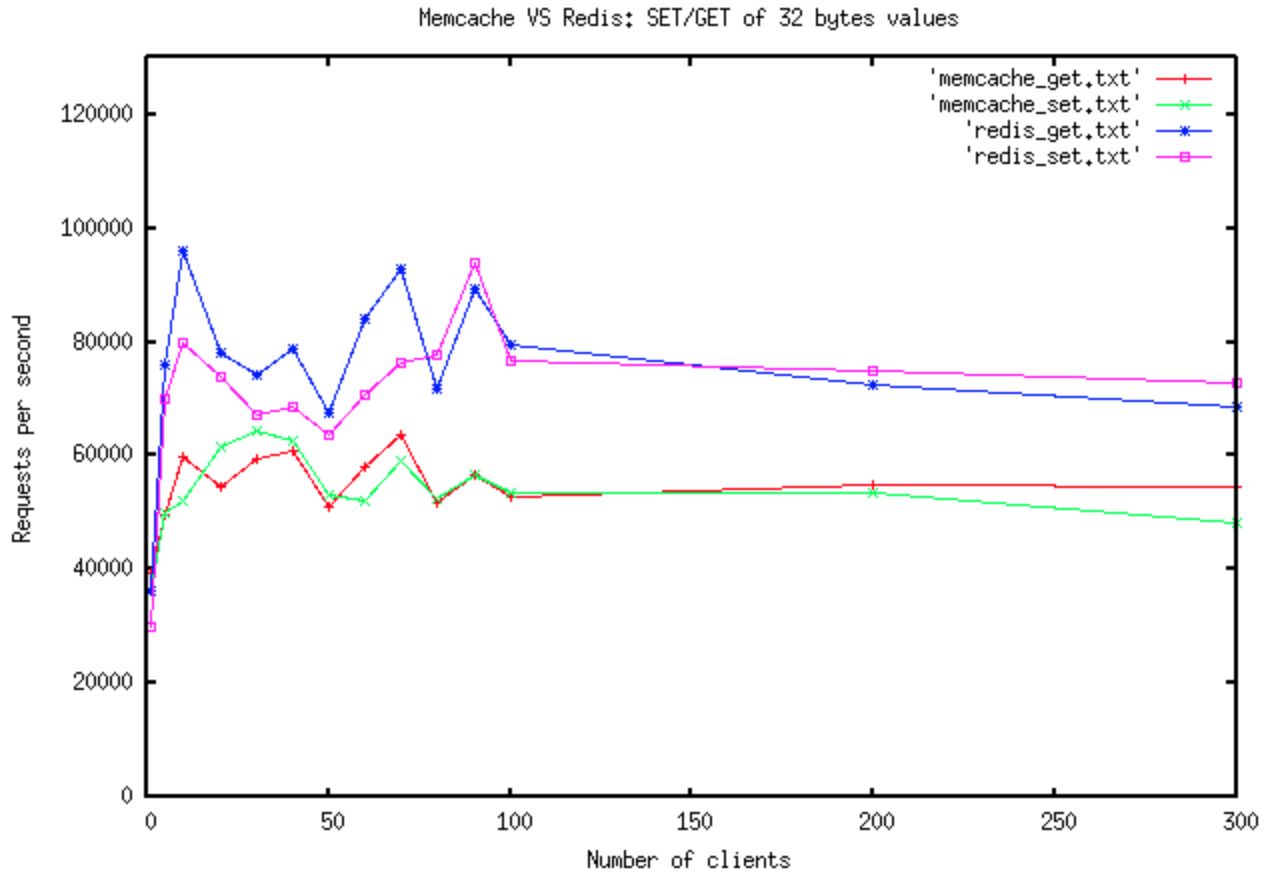
1. Security:
    i. Only 1 sign in (authentication information exchange) required for a session $\Rightarrow$ Lower the probability of hacking.
    ii. Session has TTL $\Rightarrow$ No reuse of Session ID.
2. Efficiency: No repeat authentication for every query.

Redis

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

Redis is written in ***ANSI C*** and works in most POSIX systems like Linux, *BSD, OS X without external dependencies.

Redis achieves very high throughput and scalability.



(MemCached is another popular in-memory key-value storage database.)

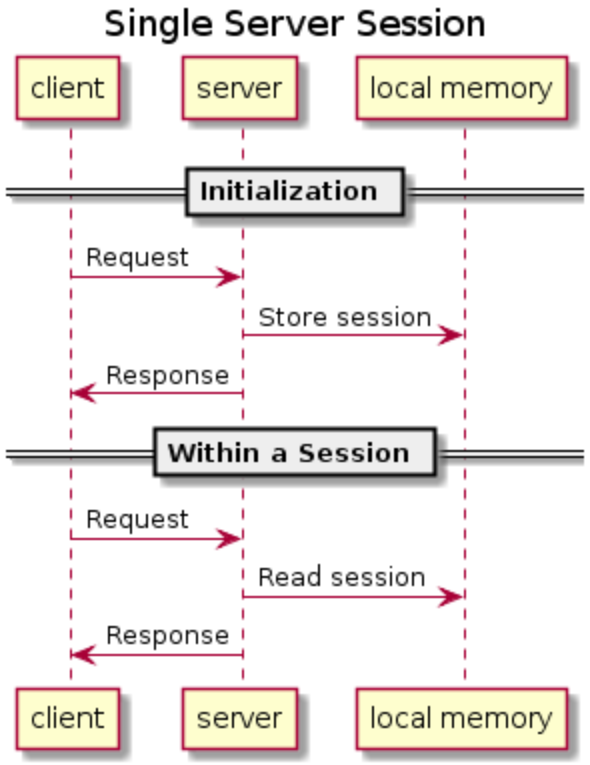
Redis supports the **cluster** mode. It can easily add one more machine to the cluster to scale out the storage. Different machines store different hash slots so there is low dependency between machines.

It also supports **Master-Slave** replication to increase the **reliability**.

Moreover, **sentinels** can be added to monitor the states of the machines in a cluster to increase the **availability**.

Applying Redis For Session Storage

There are a lot of ways to implement **Session**. For the case of a single server, **Session** can be implemented as a hash table in the local memory.

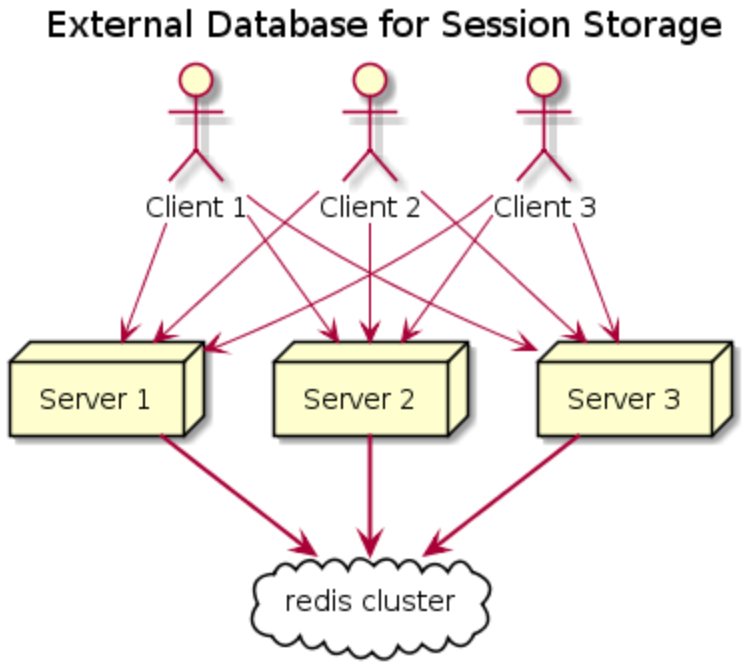


However, the problem comes when we have **multiple** application servers that share the same session. If we still use the local memory, a user may query different servers for different requests and may fail to access the previous sign in information.

Another approach is to synchronize the memory between different servers. However, memory synchronization is also expensive!

A simple approach is to use a database (cluster) for all application servers to share the session informaiton. However, the traditional relational database is relatively slow and may be a bottleneck!

⇒ We can use **in-memory** key-value (NoSQL) database to achieve high throughput for Session storage.



Problem

Redis: In-memory $\Rightarrow$ High throughput (Fast).

In-memory $\Rightarrow$ Memory Resource is limited and treasure!

When memory is used up,

1. We don't set a memory limit for ***Redis***: OS will use **swap** for more virtual memory $\Rightarrow$ Very slow!.
2. We set a memory limit $\Rightarrow$ memory *eviction*.
3. For *volatile* data, ***Redis*** also has expiration strategy to evict them.

Later we will discuss more detail about the *eviction* and *expiration* strategy.

Now, consider the case storing great amounts of **Session** data in ***Redis***. ***Redis*** does not have sepcial configuration for the use case of **Session**.

$\Rightarrow$ Can we improve the performance of ***Redis*** for **Session Storage**?

Characteristics of Session Data

Session data has special distribution and characteristics that worths extra consideration.

1. **Session data** always has **TTL**.
2. **TTL** is usually fixed, e.g. 1 hour for all sessions.

3. Unlike cache, **session data** will be loaded from the (relational) database (e.g. MySQL) and store in Redis for the whole session. It cannot be easily evicted like a cache.
   Cache: Cache miss ⇒ Load data from the database again ⇒ Efficiency issue.
   Session data: Eviction will cause the user to sign in again! ⇒ Efficiency and data lost issue.!
   For example, a session data may look like

```
{
    sessionID: {
        userID
        userName
        userInformation
    }
}
```

   where the *sessionID* is a key to access the user's other attributes.

4. TTL is **frequently** reset, because every new query will refresh the TTL.
5. **Session data** will have significantly great data flow for some scenarios, e.g. at the moment of 11-11 (双十一). And the **TTL** will also be reached simultaneously. At the same time, there will be new users come.
6. **Session data** may have correlationship with each other.

Current Approach for Redis

Expire

(https://redis.io/commands/expire)

Two ways are used to expire a key:

1. **Passive** way: When accessing an expired key, ***Redis*** deletes it.
2. **Active** way.

Active way for expiration: **periodic random** expiration:

Every second, ***Redis*** does the following for **10 times**:

1. Test **20** random keys from the set of keys with an associated expire.
2. Delete all the keys found expired.
3. If **more than 25%** of keys were expired, start again from step 1.

Theoretically, the algorihtm will result in at most **25%** of undeleted expired keys.

Here, **10 times per second**, **20 keys** and **25%** are const parameters. It is a tradeoff between CPU and memory usage.

Key Eviction For Redis

(https://redis.io/topics/lru-cache)

When the memory exceeds the memory limit, ***Redis*** supports several key eviction policies:

1. **noeviction**: Return error.
2. **allkeys-lru**
3. **volatile-lru**
4. **allkeys-random**
5. **volatile-random**
6. **volatile-ttl**: Shorter ttl first.
7. **allkeys-lfu**
8. **volatile-lfu**

Eviction Process works as follows:

1. A client runs a new command, resulting in more data added.
2. ***Redis*** checks the memory usage, and if it is greater than the maxmemory limit , it evicts keys according to the policy.
3. A new command is executed, and so forth.

Therefore, ***Redis*** may continuously *cross* the boundaries of the memory limit, $\Rightarrow$ by going over it, $\Rightarrow$ and then by *evicting* keys to return back under the limits.

Ideas

1. Keeping track of the number of "expiring" elements every second:

   Memory usage for 1 hour: $60 \times 60 = 3600\ int = 14.4KB$
   Memory usage for 1 day (**Session** length usually does not exceed 1 day): $14.4 \times 24 = 345.6KB \Rightarrow$ Relatively small amount of data.

   Redis instance without any storage data will occupy about **3MB startup memory**. (https://redis.io/topics/faq)

   Memory usage: O(1) (only related to the time length, which is a constant parameter).
   Computation complexity of mantaining the list: O(1).

2. Can we get a **better algorithm** than the **static** configuration using the information?

   Recall the current approach:

   Every second, ***Redis*** does the following for **10 times**:

   i. Test **20** random keys from the set of keys with an associated expire.
   ii. Delete all the keys found expired.
   iii. If **more than 25%** of keys were expired, start again from step 1.

3. Can we **batch** key expiration? Because **Session storage** has its special distribution!

Related Works

1. *pRedis* (SoCC, 2019): Penalty and Locality Aware **Memory Allocation** in Redis (Redis as a Cache)
2. *Redis++* (Journal of Computers, 2019): A High Performance Memory Key-Value Database Based on Redis (Memory Management, **2-level cache index** to solve collision)

Performance Evaluation

Benchmark Tools

(https://redis.io/topics/benchmarks)

Redis has several mature benchmark tools to evaluate the performance:

1. **redis-benchmark** can be used to run a set of tests for ***Redis***.
2. **memtier_benchmark** from Redis Labs is a NoSQL Redis and Memcache traffic generation and benchmarking tool.
3. **rpc-perf** from Twitter is a tool for benchmarking RPC services that supports Redis and Memcache.
4. **YCSB** from Yahoo @Yahoo is a benchmarking framework with clients to many databases, including Redis.

Redis's performance is mainly influenced by the following factors:\

1. Network bandwidth and latency ⋆.
2. CPU ⋆: Single-threaded model.
3. Speed of RAM and memory bandwidth ○.
4. VM ○.
5. Unix domain socket > TCP/IP.

Conclusion

Redis is a popular key-value **in-memory** database with high throughput and high availability. It is widely used to handle the **Session Storage** for a cluster of servers. However, ***Redis*** does not have specific algorithm or configuration for **Session Storage**, which shows the potential to optimize the performance under the usage for **Session Storage**.