Vineet Sagar Self Prep System D... Latest commit b89d37c on Apr 22, 2020 History

0 contributors

        - Client-side web storage
        - Web server cache
        - CDN
        - Scaling
    - Caching rules of thumb
        - Cache priority
        - Cache reuse
        - Cache invalidation
    - Pains
        - Pain of large data sets - When cache memory is full
        - Pain of stale data
        - Pain of loading
        - Pain of duplication
    - Thundering herd problem
        - Def
        - Solutions
    - Scaling Memcached at Facebook
- Redis
    - Data structure
        - SDS (Simple dynamic string)
        - Hash
            - Structure
            - Incremental resizing
            - Encoding
        - Skiplist
            - Skiplist vs balanced tree in ZSet
        - Memory efficient data structures
            - Ziplist
                - Structure
                - Complexity
            - IntSet
                - Structure
                - Upgrade and downgrade
        - Object

- Advanced data structures
  - HyperLogLog
  - Bloomberg filter
  - Bitmap
  - Stream
- Implement a cache system on a single machine
  - Server
  - Client
  - Processing logic
    - Event
      - File Event
      - Time event
    - I/O multi-plexing
  - Interaction modes between server and client
    - RESP
    - Pipeline
    - Transaction
    - Script mode
    - PubSub model
  - Expiration strategy
    - History
      - Types
    - Commands
    - Eviction options
  - Persistence options
    - COW
    - Pros and Cons between RDB and AOF
    - RDB

- Implement a cache system on a distributed scale
  - Consistency - Replication and failover
  - Availability - Sentinel
    - Definition
    - Advanced concepts

- Algorithms and internals
- Scalability Redis cluster
  - Main properties and rational of the design
    - Redis cluster goals
    - Implemented subset
    - Clients and Servers roles in the Redis Cluster protocol
    - Write safety
    - Availability
  - Redis cluster main components
    - Key distribution model
    - Cluster node attributes
    - Message and implementation
  - Redirection and resharding
    - Resharding condition
    - Resharding commands
    - Resharding internals
    - Move and Ask redirection
    - Smart client
  - Fault tolerance
    - Heartbeat and gossip messages
    - Failure detection
      - PFAIL to FAIL state
      - Weak agreement
      - FAIL propogation
  - Configuration handling, propogation and failovers
    - Cluster current epoch
    - Configuration epoch
    - Slave election and promotion
    - Hash slots configuration propagation
    - Replica migration
    - ConfigEpoch conflicts resolution algorithm
- Application Components: - Bulkhead
  - Distributed locking
  - Pubsub

Cache

Intuition

- Locality of reference
- Long tail

Factors for hit ratio

- Size of cache key space
  - The more unique cache keys your application generates, the less chance you have to reuse any one of them. Always consider ways to reduce the number of possible cache keys.
- The number of items you can store in cache
  - The more objects you can physically fit into your cache, the better your cache hit ratio.
- Longevity
  - How long each object can be stored in cache before expiring or being invalidated.

Applicable scenarios

- short answer

    - How many times a cached piece of data can and is reused by the application
    - the proportion of response time that is alleviated by caching
- In applications that are I/O bound, most of the response time is getting data from a database.

Access pattern

Write through cache

- def: write go through the cache and write is confirmed as success only if writes to DB and the cache both succeed.
- use-case: applications which write and re-read the information quickly. But the write latency might be much higher because of two write phase

Write around cache

- def: write directly goes to the DB. The cache reads the info from DB in case of a miss
- use-case: lower write load to cache and faster writes, but can lead to higher read latency in case of applications which write and re-read the information quickly

Write back cache

- def: write is directly done to the caching layer and write is confirmed as soon as the write to the cache completes.The cache then asynchronously syncs this write to the DB.
- use-case: quick write latency and high write throughput. But might lose data in case the cache layer dies

How to handle cache failure

- Facebook Lease Get

Types

Browser cache

- Browsers have built-in caching capabilities to reduce the number of request sent out. These usually uses a combination of memory and local files.
- There are several problems with browser cache
    - The size of the cache tends to be quite small by default. Usually around 1GB. Given that web pages have become increasingly heavy, browsers would probably be more effective if they defaulted to much larger caches.
    - When the cache becomes full, the algorithm to decide what to remove is crude. Commonly, the LRU algorithm is used to purge old items. It fails to take into account the relative "cost" to request different types of resources. For example, the loading of Javascript resources typically blocks loading of the rest of the page. It makes more sense for these to be given preference in the cache over, say, images.
    - Many browsers offer an easy way for the user to remove temporary data for the sake of privacy. Users often feel that cleaning the browser cache is an important step in somehow stopping their PC from running slow.

Caching proxies

- A caching proxy is a server, usually installed in a local corporate network or by the Internet service provider (ISP). It is a read-through cache used to reduce the amount of traffic generated by the users of the network by reusing responses between users of the network. The larger the network, the larger the potential savings - that is why it was quite common among ISPs to install transparent caching proxies and route all of the HTTP traffic through them to cache as many requests as possible.
- In recent years, the practice of installing local proxy servers has become less popular as bandwidth has become cheaper and as it becomes more popular for websiste to serve their resources soley over the Secure Socket Layer.

Reverse proxy

- A reverse proxy works in the exactly same way as a regular caching proxy,

but the intent is to place a reverse proxy in your own data center to reduce the load put on your web servers.

- Purpose:
  - For caching, they can be used to lighten load on the back-end server by serving up cached versions of dynamically generated pages (thus cuttping CPU usage). Using reverse proxies can also give you more flexibility because you can override HTTP headers and better control which requests are being cached and for how long.
  - For load balancing, they can be used for load-balancing multiple back-end web servers.

HTTP Cache

- All of the caching technologies working in the HTTP layer work as read-through caches
  - Procedures
    - First Client 1 connects to the cache and request a particular web resource.
    - Then the cache has a change to intercept the request and respond to it using a cached object.
    - Only if the cache does not have a valid cached response, will it connect to the origin server itself and forward the client's request.
  - Advantages: Read-through caches are especially attractive because they are transparent to the client. This pluggable architecture gives a lot of flexibility, allowing you to add layers of caching to the HTTP stack without needing to modify any of the clients.

Headers

- Conditional gets: If-Modified-Since header in the get request
  - If the server determines that the resource has been modified since the date given in this header, the resource is returned as normal. Otherwise, a 304 Not Modified status is returned.
  - Use case: Rather than spending time downloading the resource again, the browser can use its locally cached copy. When downloading of the resource only forms only a small fraction of the request time, it doesn't have much benefit.

- max-age inside Expires and Cache-Contrl: The resource expires on such-and-such a date. Until then, you can just use your locally cached copy.
    - The main difference is that Expires was defined in HTTP 1.0, whereas the Cache-Control family is new to HTTP 1.1. So, in theory, Expires is safer because you occasionally still encounter clients that support only HTTP 1.0. Although if both are presents, preferences are given to Cache-Control: max-age.
    - Choosing expiration policies:
        - Images, CSS, Javascript, HTML, Flash movies are primary candidates. The only type of resources you don't usually want to cache is dynamically generated content created by server-side scripting languages such as PHP, Perl and Ruby. Usually one or two months seem like a good figure.
    - Coping with stale content: There are a few tricks to make the client re-request the resource, all of which revolved around changing the URL to trick the browser into thinking the resource is not cached.
        - Use a version/revision number or date in the filename
        - Use a version/revision number or date in the path
        - Append a dummy query string
- Other headers inside Cache-Control:
    - private: The result is specific to the user who requested it and the response cannot be served to any other user. This means that only browsers will be able to cache this response because intermediate caches would not have the knowledge of what identifies a user.
    - public: Indicates the response can be shared between users as long as it has not expired. Note that you cannot specify private and public options together; the response is either public or private.
    - no-store: Indicates the response should not be stored on disks by any of the intermediate caches. In other words, the response can be cached in memory, but it will not be persisted to disk.
    - no-cache: The response should not be cache. To be accurate, it states that the cache needs to ask the server whether this response is still valid every time users request the same resource.
    - max-age: Indicates how many seconds this response can be served from the cache before becoming stale. (TTL of the response)
    - s-maxage: Indicates how many seconds this response can be served

from the cache before becoming stale on shared caches.
    - no-transformation: Indicates the response should be served without any modifications. For example, a CDN provider might transcode images to reduce their size, lowering the quality or changing the compression algorithm.
    - must-revalidate: Once the response becomes stale, it cannot be returned to clients without revalidation. Although it may seem odd, caches may return stale objects under certain conditions. For example, if the client explicitly allows it or if the cache loses connection to the original server.
- Expires:
    - Allows you to specify an absolute point in time when the object becomes stale.
    - Some of the functionality controlled by the Cache-Control header overlaps that of other HTTP headers. Expiration time of the web response can be defined either by Cache-Control: max-age=600 or by setting an absolute expiration time using the Expires header. Including both of these headers in the response is redundant and leads to confusion and potentially inconsistent behavior.
- Vary:
    - Tell caches that you may need to generate multiple variations of the response based on some HTTP request headers. For example: Vary:Accept-Encoding is the most common Vary header indicating that you may return responses encoded in different ways depending on the Accept-Encoding header that the client sends to your web server. Some clients who accept gzip encoding will get a compressed response, where others who cannot support gzip will get an uncompressed response.
- How not to cache:
    - It's common to see meta tags used in the HTML of pages to control caching. This is a poor man's cache control technique, which isn't terribly effective. Although most browsers honor these meta tags when caching locally, most intermediate proxies do not.

Content delivery networks

- A CDN is a distributed network of cache servers that work in similar way as

caching proxies. They depend on the same HTTP headers, but they are controlled by the CDN service provider.

- Advantage:
  - Reduce the load put on your servers
  - Save network bandwidth
  - Improve the user experience because by pushing content closer to your users.
- Procedures: Web applications would typically use CDN to cache their static files like images, CSS, JavaScript, videos or PDF.
  - You can imlement it easily by creating a static subdomain and generate URLs for all of your static files using this domain
  - Then you configure the CDN provider to accept these requests on your behalf and point DNS for s.example.org to the CDN provider.
  - Any time CDN fails to serve a piece of content from its cache, it forwards the request to your web servers and caches the response for subsequent users.

Scaling

- Do not worry about the scalability of browser caches or third-party proxy servers.
- This usually leaves you to manage reverse proxy servers. For most young startups, a single reverse proxy should be able to handle the incoming traffic, as both hardware reverse proxies and leading open-source ones can handle more than 10,000 requests per second from a single machine.
  - First step: To be able to scale the reverse proxy layer efficiently, you need to first focus on your cache hit ratio first.
    - Cache key space: Describe how many distinct URLs your reverse proxies will observe in a period of time. The more distinct URLs are served, the more memory or storage you need on each reverse proxy to be able to serve a significant portion of traffic from cache. Avoid caching responses that depend on the user (for example, that contain the user ID in the URL). These types of response can easily pollute your cache with objects that cannot be reused.
    - Average response TTL: Describe how long each response can be cached. The longer you cache objects, the more chance you have

to reuse them. Always try to cache objects permanently. If you cannot cache objects forever, try to negotiate the longest acceptable cache TTL with your business stakeholders.
        - Average size of cached object: Affects how much memory or storage your reverse proxies will need to store the most commonly accessed objects. Average size of cached object is the most difficult to control, but you should still keep in mind because there are some techniques that help you "shrink" your objects.
    - Second step: Deploying multiple reverse proxies in parallel and distributing traffic among them. You can also scale reverse proxies vertically by giving them more memory or switching their persistent storage to solid-state drive.

Application objects cache

- Application object caches are mostly cache-aside caches. The application needs to be aware of the existence of the object cache, and it actively uses it to store and retrieve objects rather than the cache being transparently positioned between the application and its data sources.
- All of the object cache types discussed in this section can be imagined as key-value stores with support of object expiration.

Client-side web storage

- Web storage allows a web application to use a limited amount (usually up to 5MB to 25MB of data).
- Web storage works as a key-value store.

Web server cache

- Objects are cached directly in the application's memory
- Objects are stored in shared memory segments so that multiple processes running on the same machine could access them.
- A caching server is deployed on each web server as a separate application.

CDN

Scaling

- Client-side caches like web browser storage cannot be scaled.
- The web server local caches are usually scaled by falling back to the file system.
- Distributed caches are usually scaled by data partitioning. Adding read-only slaves to sharded node.

Caching rules of thumb

Cache priority

- The higher up the call stack you can cache, the more resources you can save.
- Aggregated time spent = time spent per request * number of requests

Cache reuse

- Always try to reuse the same cached object for as many requests/users as you can.

Cache invalidation

- LRU
- TTL

Pains

Pain of large data sets - When cache memory is full

- Evict policies
  - FIFO (first-in, first out)
  - LRU (least recently used)
  - LFU (least frequently used)
  - See reference section for more discussions
- What to do with evicted one
  - Overflow to disk

    - Delete it

Pain of stale data

- Expiration policy
    - TTI: time to idle, a counter count down if not reset
    - TTL: time to leave, maximum tolerance for staleness

Pain of loading

- Persistent disk store
- Bootstrap cache loader
    - def: on startup, create background thread to pull the existing cache data from another peer
    - automatically bootstrap key on startup
    - cache value on demand

Pain of duplication

- Get failover capability but avoid excessive duplication of data
- Each node hods data it has seen
- Use load balancer to get app-level partitioning
- Use fine grained locking to get concurrency
- Use memory flush/fault to handle memory overflow and availability
- Use casual ordering to guarantee coherency

Thundering herd problem

Def

- Many readers read an empty value from the cache and subseqeuntly try to load it from the database. The result is unnecessary database load as all readers simultaneously execute the same query against the database.

- Let's say you have [lots] of webservers all hitting a single memcache key that caches the result of a slow database query, say some sort of stat for the homepage of your site. When the memcache key expires, all the webservers may think "ah, no key, I will calculate the result and save it back to memcache". Now you have [lots] of servers all doing the same expensive DB query.

```
/* read some data, check cache first, otherwise read from SoR */
public V readSomeData(K key) {
  Element element;
  if ((element = cache.get(key)) != null) {
    return element.getValue();
  }

  // note here you should decide whether your cache
  // will cache 'nulls' or not
  if (value = readDataFromDataStore(key)) != null) {
    cache.put(new Element(key, value));
  }

  return value;
}
```

Solutions

- Stale date solution: The first client to request data past the stale date is asked to refresh the data, while subsequent requests are given the stale but not-yet-expired data as if it were fresh, with the understanding that it will get refreshed in a 'reasonable' amount of time by that initial request
    - When a cache entry is known to be getting close to expiry, continue to server the cache entry while reloading it before it expires.
    - When a cache entry is based on an underlying data store and the underlying data store changes in such a way that the cache entry should be updated, either trigger an (a) update or (b) invalidation of that entry from the data store.
- Add entropy back into your system: If your system doesn't jitter then you get thundering herds.

    - For example, cache expirations. For a popular video they cache things as best they can. The most popular video they might cache for 24 hours. If everything expires at one time then every machine will calculate the expiration at the same time. This creates a thundering herd.
    - By jittering you are saying randomly expire between 18-30 hours. That prevents things from stacking up. They use this all over the place. Systems have a tendency to self synchronize as operations line up and try to destroy themselves. Fascinating to watch. You get slow disk system on one machine and everybody is waiting on a request so all of a sudden all these other requests on all these other machines are completely synchronized. This happens when you have many machines and you have many events. Each one actually removes entropy from the system so you have to add some back in.

- No expire solution: If cache items never expire then there can never be a recalculation storm. Then how do you update the data? Use cron to periodically run the calculation and populate the cache. Take the responsibility for cache maintenance out of the application space. This approach can also be used to pre-warm the the cache so a newly brought up system doesn't peg the database.
    - The problem is the solution doesn't always work. Memcached can still evict your cache item when it starts running out of memory. It uses a LRU (least recently used) policy so your cache item may not be around when a program needs it which means it will have to go without, use a local cache, or recalculate. And if we recalculate we still have the same piling on issues.
    - This approach also doesn't work well for item specific caching. It works for globally calculated items like top N posts, but it doesn't really make sense to periodically cache items for user data when the user isn't even active. I suppose you could keep an active list to get around this limitation though.

Scaling Memcached at Facebook

- In a cluster:
    - Reduce latency

        - Problem: Items are distributed across the memcached servers through consistent hashing. Thus web servers have to rountinely communicate with many memcached servers to satisfy a user request. As a result, all web servers communicate with every memcached server in a short period of time. This all-to-all communication pattern can cause incast congestion or allow a single server to become the bottleneck for many web servers.
        - Solution: Focus on the memcache client.
    - Reduce load
        - Problem: Use memcache to reduce the frequency of fetching data among more expensive paths such as database queries. Web servers fall back to these paths when the desired data is not cached.
        - Solution: Leases; Stale values;
    - Handling failures
        - Problem:
            - A small number of hosts are inaccessible due to a network or server failure.
            - A widespread outage that affects a significant percentage of the servers within the cluster.
        - Solution:
            - Small outages: Automated remediation system.
            - Gutter pool
    - In a region: Replication
    - Across regions: Consistency

Redis

- Understand the internals of the most widely used caching system

Data structure

SDS (Simple dynamic string)

- Redis implements SDS on top of c string because of the following reasons:

    i. Reduce the strlen complexity from O(n) to O(1)
    ii. Avoid buffer overflow because C needs to check string has enough capacity before executing operations such as strcat.
- SDS has the following data structure

```
struct sdshdr
{
    int len;
    int free;
    char buf[];
};
```

- SDS relies on the following two mechanisms for unused space.
    i. Space preallocation. The preallocation algorithm used is the following: every time the string is reallocated in order to hold more bytes, the actual allocation size performed is two times the minimum required. So for instance if the string currently is holding 30 bytes, and we concatenate 2 more bytes, instead of allocating 32 bytes in total SDS will allocate 64 bytes. However there is an hard limit to the allocation it can perform ahead, and is defined by SDS_MAX_PREALLOC. SDS will never allocate more than 1MB of additional space (by default, you can change this default).
    ii. Lazy free: When space is freed, it is marked as free but not immediately disallocated.
    iii. Binary safety. C structure requires char comply with ASCII standards.
    iv. Compatible with C string functions. SDS will always allocate an additional char as terminating character so that SDS could reuse some C string functions.

Hash

Structure

- dict in Redis is a wrapper on top of hashtable

```
typedef struct dict
{
    dictType *type;
```

```
    void *privdata;

    // hash table
    dictht ht[2];

    // rehash index
    // rehashing not in progress if rehashidx == -1
    int trehashidx;
}
```

Incremental resizing

- Load factor = total_elements / total_buckets
- Scale up condition: load factor >= 1 (or load factor > 5) and no heavy background process (BGSAVE or BGREWRITEAOF) is happening
- Scale down condition: load factor < 0.1
- Condition to stop rehash:
    i. Incremental hashing usually follows these conditions: dictAddRaw / dictGenericDelete / dictFind / dictGetRandomKey
    ii. Incremental hashing is also scheduled in server cron job.
- During the resizing process, all add / update / remove operations need to be performed on two tables.

Encoding

Skiplist

Skiplist vs balanced tree in ZSet

- They are not very memory intensive. It's up to you basically. Changing parameters about the probability of a node to have a given number of levels will make then less memory intensive than btrees.
- A sorted set is often target of many ZRANGE or ZREVRANGE operations, that is, traversing the skip list as a linked list. With this operation the cache locality of skip lists is at least as good as with other kind of balanced trees.
- They are simpler to implement, debug, and so forth. For instance thanks to the skip list simplicity I received a patch (already in Redis master) with augmented skip lists implementing ZRANK in O(log(N)). It required little changes to the code.

- https://github.com/antirez/redis/blob/90a6f7fc98df849a9890ab6e0da4485457bf60cd/src/ziplist.c

Memory efficient data structures

Ziplist

Structure

- zlbytes: Is a 4 byte unsigned integer, used to store the entire ziplist number of bytes used.
- zltail: Is a 4 byte unsigned integer, used to store the ziplist of the last node relative to the ziplist first address offset.
- zllen: Is a 2 byte unsigned integer, the number of nodes stored in ziplist, maximum value for (2^16 - 2), when zllen is greater than the maximum number of value when, need to traverse the whole ziplist to obtain the ziplist node.
- zlend: Is a 1 byte unsigned integer, value 255 (11111111), as the end of the ziplist match.
- entryX: Node ziplist, each node could represent a length-limited int or char.
  i. prev_entry_bytes_length:
  ii. content:

Complexity

- Insert operation. Worst case: O(N^2). Best case: O(1). Average case o(N)
    - Cascade update: When an entry is inserted, we need to set the prevlen field of the next entry to equal the length of the inserted entry. It can occur that this length cannot be encoded in 1 byte and the next entry needs to be grow a bit larger to hold the 5-byte encoded prevlen. This can be done for free, because this only happens when an entry is already being inserted (which causes a realloc and memmove). However, encoding the prevlen may require that this entry is grown as well. This effect may cascade throughout the ziplist when there are consecutive entries with a size close to ZIP_BIGLEN, so we need to check that the prevlen can be encoded in every consecutive

entry.

- Delete operation. Worst case: O(N^2). Best case: O(1). Average case o(N)
    - Cascade update: Note that this effect can also happen in reverse, where the bytes required to encode the prevlen field can shrink. This effect is deliberately ignored, because it can cause a flapping effect where a chain prevlen fields is first grown and then shrunk again after consecutive inserts. Rather, the field is allowed to stay larger than necessary, because a large prevlenfield implies the ziplist is holding large entries anyway.
- Iterate operation.
- https://redisbook.readthedocs.io/en/latest/compress-datastruct/ziplist.html

IntSet

Structure

```
typedef struct intset
{
    uint32_t encoding; // INSET_ENC_INT16, INTSET_ENC_INT32,
INTSET_ENC_INT64
    uint32_t length;
    int8_t contents[];
}
```

Upgrade and downgrade

- As long as there is one item in the content which has bigger size, the entire content array will be upgraded.
- No downgrade is provided.

Object

- Definition

```
typedef struct redisObject
```

```
{
    unsigned type:4;
    unsigned encoding:4;
    void *ptr;
} robj;
```

- Type and encoding. Encoding gives Type the flexibility to use differnt object type under different scenarios.

type	encoding
Redis_String	REDIS_ENCODING_INT
Redis_String	REDIS_ENCODING_EMBSTR
Redis_String	REDIS_ENCODING_RAW
Redis_List	REDIS_ENCODING_ZIPLIST
Redis_List	REDIS_ENCODING_LINKEDLIST
Redis_Hash	REDIS_ENCODING_ZIPLIST
Redis_Hash	REDIS_ENCODING_HT
Redis_Set	REDIS_ENCODING_INTSET
Redis_Set	REDIS_ENCODING_HT
Redis_ZSet	REDIS_ENCODING_ZIPLIST
Redis_ZSet	REDIS_ENCODING_SKIPLIST

- string
    - Three coding formats:
        a. Int: if the target could be represented using a long.
        b. Embstr: If the target is smaller than 44 bytes. Embstr is read-only but it only needs one-time to allocate free space. Embstr could also better utilizes local-cache. Represented using SDS.
        c. Raw: Longer than 45 bytes. Represented using SDS.
- Data structure to be memory efficient (https://redis.io/topics/memory-optimization)

```
hash-max-zipmap-entries 512 (hash-max-ziplist-entries for Redis
>= 2.6)
hash-max-zipmap-value 64  (hash-max-ziplist-value for Redis >=
2.6)
list-max-ziplist-entries 512
list-max-ziplist-value 64
zset-max-ziplist-entries 128
zset-max-ziplist-value 64
set-max-intset-entries 512
```

Advanced data structures

HyperLogLog

- pfadd/pfcount/pfmerge
- pf means Philippe Flajolet

Bloomberg filter

- bf.add/bf.exists/bf.madd/bf.mexists

Bitmap

- Commands: setbit/getbit/bitcountt/bitpos/bitfield

Stream

Implement a cache system on a single machine

Server

Client

Processing logic

Event

File Event

Time event

I/O multi-plexing

Interaction modes between server and client

RESP

Pipeline

- Definition: A Request/Response server can be implemented so that it is able to process new requests even if the client didn't already read the old responses. This way it is possible to send multiple commands to the server without waiting for the replies at all, and finally read the replies in a single step.
- Benefits:
  - Reduce latency because many round trip times are avoided
  - Improves a huge amount of total operations you could perform per second on a single redis server because it avoids many context switch. From the point of view of doing the socket I/O, this involves calling the read() and write() syscall, that means going from user land to kernel land. The context switch is a huge speed penalty. More detailed explanation: (why a busy loops are slow even on the loopback interface?) processes in a system are not always running, actually it is the kernel scheduler that let the process run, so what happens is that, for instance, the benchmark is allowed to run, reads the reply from the Redis server (related to the last command executed), and writes a new command. The command is now in the loopback interface buffer, but in order to be read by the server, the kernel should schedule the server process (currently blocked in a system call) to run, and so forth. So in practical terms the loopback interface still involves network-alike latency, because of how the kernel scheduler works.
  - https://redis.io/topics/pipelining
- Usage:
  - Under transaction commands such as MULTI, EXEC
  - The commands which take multiple arguments: MGET, MSET, HMGET, HMSET, RPUSH/LPUSH, SADD, ZADD
  - https://redislabs.com/ebook/part-2-core-concepts/chapter-4-

keeping-data-safe-and-ensuring-performance/4-5-non-transactional-pipelines/

- Internal: Pipeline is purely a client-side implementation.
  - Buffer the redis commands/operations on the client side
  - Synchronously or asynchronously flush the buffer periodically depending on the client library implementation.
  - Redis executes these operations as soon as they are received at the server side. Subsequent redis commands are sent without waiting for the response of the previous commands. Meanwhile, the client is generally programmed to return a constant string for every operation in the sequence as an immediate response
  - The tricky part: the final responses. Very often it is wrongly interpreted that all the responses are always read at one shot and that the responses maybe completely buffered on server's side. Even though the response to all the operations seem to arrive at one shot when the client closes the pipeline, it is actually partially buffered on both client and the server. Again, this depends on the client implementation. There does exist a possibility that the client reads the buffered response periodically to avoid a huge accumulation on the server side. But it is also possible that it doesn't. For example: the current implementation of the Jedis client library reads all responses of a pipeline sequence at once.
  - http://blog.nachivpn.me/2014/11/redis-pipeline-explained.html
- Pipeline vs transaction:
  - Whether the commands are executed atomically
- How does stackexchange implements pipelines:
  - https://stackexchange.github.io/StackExchange.Redis/PipelinesMultiplexers.html

Transaction

Script mode

PubSub model

Expiration strategy

History

- Lazy free
  - Timer function and perform the eviction. Difficulties: Adaptive speed for freeing memory. Found an adaptive strategy based on the following two standards: 1. Check the memory tendency: it is raising or lowering? In order to adapt how aggressively to free. 2. Also adapt the timer frequency itself based on “1”, so that we don’t waste CPU time when there is little to free, with continuous interruptions of the event loop. At the same time the timer could reach ~300 HZ when really needed.
  - For the above strategy, during busy times it only serves 65% QPS. However, the internal Redis design is heavily geared towards sharing objects around. Many data structures within Redis are based on the shared object structure robj. As an effort, the author changed it to SDS.
- http://antirez.com/news/93

Types

- Timing deletion: While setting the expiration time of the key, create a timer. Let the timer immediately perform the deletion of the key when the expiration time of the key comes.
- Inert deletion: Let the key expire regardless, but every time the key is retrieved from the key space, check whether the key is expired, if it is expired, delete the key; if it is not expired, return the key.
- Periodic deletion: Every once in a while, the program checks the database and deletes the expired keys. How many expired keys to delete and how many databases to check are determined by the algorithm.
- Inert deletion and periodic deletion are used within Redis.
  - Inert deletion (expireIfNeeded function) serves as a filter before executing any key operation
  - Periodic deletion of key occurs in Redis’s periodic execution task (server Cron, default every 100ms), and is the master node where Redis occurs, because slave nodes synchronize to delete key through the DEL command of the primary node. Each DB is traversed in turn (the default configuration number is 16). For each db, 20 keys (ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP) are selected

randomly for each cycle to determine whether they are expired. If the selected keys in a round are less than 25% expired, the iteration is terminated. In addition, if the time limit is exceeded, the process of expired deletion is terminated.

- https://developpaper.com/an-in-depth-explanation-of-key-expiration-deletion-strategy-in-redis/

Commands

- Proactive commands: Unlink, FlushAll Async, FlushDB Async
- Reactive commands:
  - Slve-lazy-flush: Clear data options after slave receives RDB files
  - Lazy free-lazy-eviction: full memory ejection option
  - Lazy free-lazy-expire: expired key deletion option
  - Lazyfree-lazy-server-del: Internal deletion options, such as rename oldkey new key, need to be deleted if new key exists

Eviction options

- LRU vs LFU
- https://redis.io/topics/lru-cache
- Improve LRU cache algorithm http://antirez.com/news/109

Persistence options

COW

- Both RDB and AOF relies on Unix Copy on Write mechanism
- http://oldblog.antirez.com/post/a-few-key-problems-in-redis-persistence.html

Pros and Cons between RDB and AOF

- https://redis.io/topics/persistence

RDB

- Command: SAVE vs BGSAVE. Whether a child process is forked to create RDB file.

- BGSAVE - Automatic save condition
  - saveparam format: save seconds changes
  - dirty attribute: How many databse operations have been performed after the last time.
  - lastsave: A unix timestamp - the last time the server executes SAVE or BGSAVE.

```
def serverCron():
    for saveParam in server.saveparams:
        save_internal = unixtime_now() - server.lastsave

        if server.dirty >= saveparam.changes
            and save_internal > saveparams.seconds:
                BGSAVE()
```

AOF

- FlushAppendOnlyFile's behavior depends on appendfsync param:
  - always: Write aof_buf to AOF file and sync to slaves on each file event loop.
  - everysec: Write aof_buf to AOF file on each file event loop. If the last synchronization happens before 1 sec, then sync to slaves on each file event loop.
  - no: Write aof_buf to AOF file on each file event loop. Depend on the OS to determine when to sync.
- AOF rewrite:
  - Goal: Reduce the size of AOF file.
  - AOF rewrite doesn't need to read the original AOF file. It directly reads from database.
  - Redis fork a child process to execute AOF rewrite dedicatedly. Redis opens a AOF rewrite buffer to keep all the instructions received during the rewriting process. At the end of rewriting AOF file, all instructions within AOF rewrite buffer will be flushed to the new AOF file.

Implement a cache system on a distributed scale

Consistency - Replication and failover

Availability - Sentinel

Definition

- Sentinel is Redis' resiliency solution to standalone redis instance.
  - See Compare redis deployments for details.
- It will monitor your master & slave instances, notify you about changed behaviour, handle automatic failover in case a master is down and act as a configuration provider, so your clients can find the current master instance.
- Redis sentinel was born in 2012 and first released when Redis 2.4 was stable.

Advanced concepts

- Initialization: Sentinel is a redis server running on a special mode
  - Sentinel will not load RDB or AOF file.
  - Sentinel will load a special set of Sentinel commands.
- Downstate
  - Subjective downstate: An SDOWN condition is reached when it does not receive a valid reply to PING requests for the number of seconds specified in the configuration as is-master-down-after-milliseconds parameter.
  - Objectively downstate: When enough Sentinels (at least the number configured as the quorum parameter of the monitored master) have an SDOWN condition, and get feedback from other Sentinels using the SENTINEL is-master-down-by-addr command.
- Sentinels and slaves auto discovery
  - You don't need to configure a list of other Sentinel addresses in every Sentinel instance you run, as Sentinel uses the Redis instances Pub/Sub capabilities in order to discover the other Sentinels that are monitoring the same masters and slaves. Similarly you don't need to configure what is the list of the slaves attached to a master, as Sentinel will auto discover this list querying Redis.
  - Process
    - Every Sentinel publishes a message to every monitored master and slave Pub/Sub channel **sentinel**:hello, every two seconds,

announcing its presence with ip, port, runid.
        - Every Sentinel is subscribed to the Pub/Sub channel **sentinel**:hello of every master and slave, looking for unknown sentinels. When new sentinels are detected, they are added as sentinels of this master.
        - Hello messages also include the full current configuration of the master. If the receiving Sentinel has a configuration for a given master which is older than the one received, it updates to the new configuration immediately.
- Slave selection: Relies on Raft protocol
    - The slave selection process evaluates the following information about slaves:
        - Disconnection time from the master.
        - Slave priority.
        - Replication offset processed.
        - Run ID.
    - A slave that is found to be disconnected from the master for more than ten times the configured master timeout (down-after-milliseconds option), plus the time the master is also not available from the point of view of the Sentinel doing the failover, is considered to be not suitable for the failover and is skipped.
    - The slave selection only considers the slaves that passed the above test, and sorts it based on the above criteria, in the following order.
        - The slaves are sorted by slave-priority as configured in the redis.conf file of the Redis instance. A lower priority will be preferred.
        - If the priority is the same, the replication offset processed by the slave is checked, and the slave that received more data from the master is selected.
        - If multiple slaves have the same priority and processed the same data from the master, a further check is performed, selecting the slave with the lexicographically smaller run ID. Having a lower run ID is not a real advantage for a slave, but is useful in order to make the process of slave selection more deterministic, instead of resorting to select a random slave.
- Sentinel's server cron operations:

    - Detect instance's objective and subjective downstate by sending PING commands
    - Automatically discover sentinel and slave nodes by subscribing to channel **sentinel**:hello
    - Leader selection and failover

Algorithms and internals

- Quorum
    - Use cases: Considering a master as objectively downstate; Authorizing the failover process
    - Quorum could be used to tune sentinel in two ways:
        - If a the quorum is set to a value smaller than the majority of Sentinels we deploy, we are basically making Sentinel more sensible to master failures, triggering a failover as soon as even just a minority of Sentinels is no longer able to talk with the master.
        - If a quorum is set to a value greater than the majority of Sentinels, we are making Sentinel able to failover only when there are a very large number (larger than majority) of well connected Sentinels which agree about the master being down.
- Configuratin epochs
    - Epoch is similar to Raft algorithm's term.
    - When a Sentinel is authorized, it gets a unique configuration epoch for the master it is failing over. This is a number that will be used to version the new configuration after the failover is completed. Because a majority agreed that a given version was assigned to a given Sentinel, no other Sentinel will be able to use it.
- https://redis.io/topics/sentinel

Scalability Redis cluster

Main properties and rational of the design

Redis cluster goals

- High performance and linear scalability to 1000 nodes
    - No proxies

    - Asynchronous replication
    - No merge operations on values
- Acceptable degree of write safety
    - The system tries to retain all writes originating from clients
- Availability:
    - Redis Cluster is able to survive partitions where the majority of the master nodes are reachable and there is at least one reachable slave for every master node that is no longer reachable.
    - Moreover using replicas migration, masters no longer replicated by any slave will receive one from a master which is covered by multiple slaves.

Implemented subset

- Same as standalone redis: Single key operations and multi-key operations
- Not supported: multi-database operations such as SELECT
- Additional supported: Hashtag which forces keys to be stored in the same slot

Clients and Servers roles in the Redis Cluster protocol

- Server: Holding the data, taking state of the cluster (Including mapping keys to the correct shard), auto-discover nodes, detect non-working nodes and promoting slaves to master when needed
    - Servers communicate to each other using a TCP bus and a binary protocol, called Redis Cluster Bus.
- Client: The clients is free to send requests to any node within the cluster.

Write safety

- There are two cases where write will fail
    - Clients write to master. While the master may be able to reply to the clients, the write may not be propogated to slaves and the master dies. As a result of master becomes unreacheable beyond a fixed amount of time, one of its slave is promoted.
    - A master becomes unavailable because of network partitions and failed over by ones of its slaves. The clients continues talking to the old master. However, it is unlikely to happen because:

a. Master which fails to communicate with majority of nodes will reject writes and after the partition heals writes are still rejected to allow other nodes informing configuration changes.
b. The clients haven't updated its routing table.
c. Writes to the minority of a cluster has a higher chance of being lost.

Availability

- Redis Cluster is designed to survive failures of a few nodes in the cluster, but it is not a suitable solution for applications that require availability in the event of large net splits.

Redis cluster main components

Key distribution model

- HASH_SLOT = CRC16(key) mod 16384
- Hashtag could force multiple keys are allocated in the same slots and it is used to implement multi-key operations in redis cluster ???

Cluster node attributes

- The node ID, IP and port of the node, a set of flags, what is the master of the node if it is flagged as slave, last time the node was pinged and the last time the pong was received, the current configuration epoch of the node (explained later in this specification), the link state and finally the set of hash slots served.
- https://redis.io/commands/cluster-nodes

Message and implementation

- MEET/PING/PONG: Implemented using Gossip protocol. ???
- FAIL: Broadcast because Gossip Protocol takes time.
- PUBLISH: When client sends a Publish command to the node, the node will publish this message to the channel.

Redirection and resharding

Resharding condition

- To add a new node to the cluster an empty node is added to the cluster and some set of hash slots are moved from existing nodes to the new node.
- To remove a node from the cluster the hash slots assigned to that node are moved to other existing nodes.
- To rebalance the cluster a given set of hash slots are moved between nodes.
- All the above three conditions could be abstracted as moving slots between different shards.

Resharding commands

- CLUSTER ADDSLOTS slot1 [slot2] ... [slotN]
- CLUSTER DELSLOTS slot1 [slot2] ... [slotN]
- CLUSTER SETSLOT slot NODE node
- CLUSTER SETSLOT slot MIGRATING node
- CLUSTER SETSLOT slot IMPORTING node

Resharding internals

1. redis-trib sends target node CLUSTER SETSLOT $slot IMPORTING $source_id so that target node is prepared to import key value pairs from slot.
    - On the node side, there is a bitmap

```
typedef struct clusterState
{
    // ...
    clusterNode *importing_slots_from[16384];

    // ...
}
```

2. redis-trib sends source node CLUSTER SETSLOT $slot MIGRATING $target_id so that source node is prepared to migrate key value pairs to slot.
    - On the node side, there is a bitmap

```
typedef struct clusterState
{
    // ...
    clusterNode *migrating_slots_to[16384];

    // ...
}
```

3. redis-trib sends source node CLUSTER GETKEYSINSLOT $slot $count to get at most count number of key names belonging to slot.
4. for every key name obtained in step 3, redis-trib will send source node a MIGRATE $target_ip $target_port $key_name 0 $time_out command to migrate the slots from source to dest node.
5. Repeat step 3 and 4 until all key-value pairs belong to the slots have been migrated.
6. redis-trib sends CLUSTER SETSLOT $slot NODE $target_id which will be broadcasted to all the nodes within the cluster.

Move and Ask redirection

- MOVED means that we think the hash slot is permanently served by a different node and the next queries should be tried against the specified node, ASK means to send only the next query to the specified node.
- ASK semantics for client:
  - If ASK redirection is received, send only the query that was redirected to the specified node but continue sending subsequent queries to the old node.
  - Start the redirected query with the ASKING command.
  - Don't yet update local client tables to map hash slot 8 to B.
- ASK semantics for server:
  - If the client has flag REDIS_ASKING and clusterStates_importing_slots_from[i] shows node is importing key value i, then node will execute the the client command once.

Smart client

- Redis Cluster clients should try to be smart enough to memorize the slots configuration. However this configuration is not required to be up to date.

Since contacting the wrong node will simply result in a redirection, that should trigger an update of the client view.

- Clients usually need to fetch a complete list of slots and mapped node addresses in two different situations:
  - At startup in order to populate the initial slots configuration.
  - When a MOVED redirection is received.

Fault tolerance

Heartbeat and gossip messages

- Usually nodes send ping packets that will trigger the receivers to reply with pong packets. However this is not necessarily true.
- Every node makes sure to ping every other node that hasn't sent a ping or received a pong for longer than half the NODE_TIMEOUT time.
  - For example in a 100 node cluster with a node timeout set to 60 seconds, every node will try to send 99 pings every 30 seconds, with a total amount of pings of 3.3 per second. Multiplied by 100 nodes, this is 330 pings per second in the total cluster.

Failure detection

- When a PFAIL could be escalated to FAIL
- The conditions when FAIL conditions could be cleared.
- Eventually all the nodes should agree about the state of a given node.

PFAIL to FAIL state

- PFAIL: PFAIL means Possible failure, and is a non-acknowledged failure type. A node flags another node with the PFAIL flag when the node is not reachable for more than NODE_TIMEOUT time. Both master and slave nodes can flag another node as PFAIL, regardless of its type.
- FAIL: FAIL means that a node is failing and that this condition was confirmed by a majority of masters within a fixed amount of time.
- PFAIL => FAIL: A PFAIL condition is escalated to a FAIL condition when the following set of conditions are met:
  - Some node, that we'll call A, has another node B flagged as PFAIL.
  - Node A collected, via gossip sections, information about the state of B

from the point of view of the majority of masters in the cluster.
    - The majority of masters signaled the PFAIL or FAIL condition within NODE_TIMEOUT * FAIL_REPORT_VALIDITY_MULT time. (The validity factor is set to 2 in the current implementation, so this is just two times the NODE_TIMEOUT time).
- FAIL => Normal: FAIL flag can only be cleared in the following situations:
    - The node is already reachable and is a slave. In this case the FAIL flag can be cleared as slaves are not failed over.
    - The node is already reachable and is a master not serving any slot. In this case the FAIL flag can be cleared as masters without slots do not really participate in the cluster and are waiting to be configured in order to join the cluster.
    - The node is already reachable and is a master, but a long time (N times the NODE_TIMEOUT) has elapsed without any detectable slave promotion. It's better for it to rejoin the cluster and continue in this case.

Weak agreement

- PFAIL => FAIL is a week agreement because:
    - Nodes collect views of other nodes over some time period, so even if the majority of master nodes need to agree, actually this is just state that we collected from different nodes at different times and we are not sure, nor we require, that at a given moment the majority of masters agreed.
    - While every node detecting the FAIL condition will force that condition on other nodes in the cluster using the FAIL message, there is no way to ensure the message will reach all the nodes. For instance a node may detect the FAIL condition and because of a partition will not be able to reach any other node.
- PFAIL => FAIL is an eventually consistency agreement because:
    - Eventually all the nodes should agree about the state of a given node. There are two cases that can originate from split brain conditions. Either some minority of nodes believe the node is in FAIL state, or a minority of nodes believe the node is not in FAIL state. In both the cases eventually the cluster will have a single view of the state of a given node:

    - Case 1: If a majority of masters have flagged a node as FAIL, because of failure detection and the chain effect it generates, every other node will eventually flag the master as FAIL, since in the specified window of time enough failures will be reported.
    - Case 2: When only a minority of masters have flagged a node as FAIL, the slave promotion will not happen (as it uses a more formal algorithm that makes sure everybody knows about the promotion eventually) and every node will clear the FAIL state as per the FAIL state clearing rules above (i.e. no promotion after N times the NODE_TIMEOUT has elapsed).

FAIL propogation

- The FAIL message will force every receiving node to mark the node in FAIL state, whether or not it already flagged the node in PFAIL state.

Configuration handling, propogation and failovers

Cluster current epoch

- currentEpoch lifetime
    - At node creation every Redis Cluster node, both slaves and master nodes, set the currentEpoch to 0.
    - Every time a packet is received from another node, if the epoch of the sender (part of the cluster bus messages header) is greater than the local node epoch, the currentEpoch is updated to the sender epoch.
    - Because of these semantics, eventually all the nodes will agree to the greatest configEpoch in the cluster.
- currentEpoch use case
    - Currently this happens only during slave promotion, as described in the next section. Basically the epoch is a logical clock for the cluster and dictates that given information wins over one with a smaller epoch.

Configuration epoch

- configEpoch lifetime
    - The configEpoch is set to zero in masters when a new node is created.
    - A new configEpoch is created during slave election. Slaves trying to

replace failing masters increment their epoch and try to get authorization from a majority of masters. When a slave is authorized, a new unique configEpoch is created and the slave turns into a master using the new configEpoch.

- configEpoch use case
  - configEpoch helps to resolve conflicts when different nodes claim divergent configurations (a condition that may happen because of network partitions and node failures).

Slave election and promotion

1. Condition to start the election
   - The slave's master is in FAIL state. As soon as a master is in FAIL state, a slave waits a short period of time before trying to get elected. That delay is computed as follows:
     - DELAY = 500 milliseconds + random delay between 0 and 500 milliseconds + SLAVE_RANK * 1000 milliseconds.
     - The fixed delay ensures that we wait for the FAIL state to propagate across the cluster, otherwise the slave may try to get elected while the masters are still unaware of the FAIL state, refusing to grant their vote.
     - The random delay is used to desynchronize slaves so they're unlikely to start an election at the same time.
     - The SLAVE_RANK is the rank of this slave regarding the amount of replication data it has processed from the master. Slaves exchange messages when the master is failing in order to establish a (best effort) rank: the slave with the most updated replication offset is at rank 0, the second most updated at rank 1, and so forth. In this way the most updated slaves try to get elected before others.
   - The master was serving a non-zero number of slots.
   - The slave replication link was disconnected from the master for no longer than a given amount of time, in order to ensure the promoted slave's data is reasonably fresh. This time is user configurable.
2. A slave increments its currentEpoch counter, and requests votes from master instances.
   - Votes are requested by the slave by broadcasting a

FAILOVER_AUTH_REQUEST packet to every master node of the cluster. Then it waits for a maximum time of two times the NODE_TIMEOUT for replies to arrive (but always for at least 2 seconds).

- Once the slave receives ACKs from the majority of masters, it wins the election. Otherwise if the majority is not reached within the period of two times NODE_TIMEOUT (but always at least 2 seconds), the election is aborted and a new one will be tried again after NODE_TIMEOUT * 4 (and always at least 4 seconds).

3. A master grant the vote if the following conditions are met
   - A master only votes a single time for a given epoch, and refuses to vote for older epochs: every master has a lastVoteEpoch field and will refuse to vote again as long as the currentEpoch in the auth request packet is not greater than the lastVoteEpoch. When a master replies positively to a vote request, the lastVoteEpoch is updated accordingly, and safely stored on disk.
   - A master votes for a slave only if the slave's master is flagged as FAIL.
   - Auth requests with a currentEpoch that is less than the master currentEpoch are ignored. Because of this the master reply will always have the same currentEpoch as the auth request. If the same slave asks again to be voted, incrementing the currentEpoch, it is guaranteed that an old delayed reply from the master can not be accepted for the new vote.
4. Once a master has voted for a given slave, replying positively with a FAILOVER_AUTH_ACK, it can no longer vote for another slave of the same master for a period of NODE_TIMEOUT * 2. In this period it will not be able to reply to other authorization requests for the same master.
   - A slave discards any AUTH_ACK replies with an epoch that is less than the currentEpoch at the time the vote request was sent. This ensures it doesn't count votes intended for a previous election.
5. Once a slave wins the election, it obtains a new unique and incremental configEpoch which is higher than that of any other existing master. It starts advertising itself as master in ping and pong packets, providing the set of served slots with a configEpoch that will win over the past ones.

Hash slots configuration propagation

- Two types of messages
  - Heartbeat messages: The sender of a ping or pong packet always adds information about the set of hash slots it (or its master, if it is a slave) serves.
  - Update messages: Since in every heartbeat packet there is information about the sender configEpoch and set of hash slots served, if a receiver of a heartbeat packet finds the sender information is stale, it will send a packet with new information, forcing the stale node to update its info.
- Rules to update configuration
  - Rule 1: If a hash slot is unassigned (set to NULL), and a known node claims it, I'll modify my hash slot table and associate the claimed hash slots to it.
  - Rule 2: If a hash slot is already assigned, and a known node is advertising it using a configEpoch that is greater than the configEpoch of the master currently associated with the slot, I'll rebind the hash slot to the new node.
- Rules to choose configuration
  - So the actual Redis Cluster node role switch rule is: A master node will change its configuration to replicate (be a slave of) the node that stole its last hash slot.

Replica migration

- Def: Replica migration is the process of automatic reconfiguration of a slave in order to migrate to a master that has no longer coverage (no working slaves).
- Algorithm:
  i. To start we need to define what is a good slave in this context: a good slave is a slave not in FAIL state from the point of view of a given node.
  ii. The execution of the algorithm is triggered in every slave that detects that there is at least a single master without good slaves.
    - However among all the slaves detecting this condition, only a subset should act. The subset is usually a single slave. The acting slave is the slave among the masters with the maximum number of attached slaves, that is not in FAIL state and has the smallest node ID.

iii. If the race happens in a way that will leave the ceding master without slaves, as soon as the cluster is stable again the algorithm will be re-executed again and will migrate a slave back to the original master.
iv. The algorithm is controlled by a user-configurable parameter called cluster-migration-barrier: the number of good slaves a master must be left with before a slave can migrate away. For example, if this parameter is set to 2, a slave can try to migrate only if its master remains with two working slaves.

ConfigEpoch conflicts resolution algorithm

1. IF a master node detects another master node is advertising itself with the same configEpoch.
2. AND IF the node has a lexicographically smaller Node ID compared to the other node claiming the same configEpoch.
3. THEN it increments its currentEpoch by 1, and uses it as the new configEpoch.

Application Components:

Bulkhead

- Thread Isolation: The standard approach is to hand over all requests to component C to a separate thread pool with a fixed number of threads and no (or a small) request queue.
    - Drawbacks: The primary drawback of thread pools is that they add computational overhead. Each command execution involves the queueing, scheduling, and context switching involved in running a command on a separate thread.
        - Costs of threads: At the 90th percentile there is a cost of 3ms for having a separate thread; At the 99th percentile there is a cost of 9ms for having a separate thread.
    - Advantages: The advantage of the thread pool approach is that requests that are passed to C can be timed out, something that is not possible when using semaphores.

- Semaphore Isolation: The other approach is to have all callers acquire a permit (with 0 timeout) before requests to C. If a permit can't be acquired from the semaphore, calls to C are not passed through.
- Further references
    - https://stackoverflow.com/questions/34519/what-is-a-semaphore
    - A little book about semaphore
    - https://github.com/Netflix/Hystrix/wiki/How-it-Works

Distributed locking

- Please see Distributed lock

Pubsub

Properties of pubsub

- Pub/Sub works under the premise of “fire and forget”. This essentially means that every published message will be delivered to as many subscribers as there are then it will be lost from the buffer
- All messages will be delivered to all subscribers. Mind you, you can have subscribers listening for different channels, which would prevent this from happening. But if you have more than one subscriber on the same channel, then all of them would get the same message. It would be up to them then, to decide what to do about that.
- There is no ACK message. Some communication protocols deal with an acknowledge message, in order for the subscribers to let the publisher know the message was received. In this case, there is nothing like that, so if your subscriber gets the message and then crashes, that data will be lost for good.

When to use Pubsub

- Chat servers, allowing you to create chat rooms easily by letting Redis take care of all the hard work of distributing messages amongst users. By default, these chat rooms would not persist messages, but you could find a way around that by adding some storage logic to your chat server
- Notification service: Another interesting use case, where you can

subscribe to a set of notifications you'd like to receive, and then it's a matter of publishers sending them to the right channel

- Log centralization. You could easily build a logging hub, where your own app is the publisher and different services make sure they send the information to the right destination. This would allow you to have a very flexible logging scheme, being able to swap from storing to disk to sending everything to an ELK instance or to a cloud service, or even all of them at once! Think about the possibilities!

Blocklist vs Pubsub

- Messages aren't distributed to all subscribers, in fact, every message is only delivered to one subscriber thanks to the fact that the first one to be notified, pops it out
- The fact that messages are stored in a list in Redis, they are stored inside it until a subscriber is connected. And if you configure Redis to store data in the disk, you can get a pretty reliable queueing system

Stream

- Add data to stream: Because streams are an append only data structure, the fundamental write command, called XADD, appends a new entry into the specified stream. A stream entry is not just a string, but is instead composed of one or multiple field-value pairs.
- Get data from stream:
  - Access mode 1: With streams we want that multiple consumers can see the new messages appended to the Stream, like many tail -f processes can see what is added to a log. Using the traditional terminology we want the streams to be able to fan out messages to multiple clients.
  - Access mode 2: Get messages by ranges of time, or alternatively to iterate the messages using a cursor to incrementally check all the history.
  - Access mode 3: as a stream of messages that can be partitioned to multiple consumers that are processing such messages, so that groups of consumers can only see a subset of the messages arriving in a single stream. In this way, it is possible to scale the message processing across different consumers, without single consumers

having to process all the messages: each consumer will just get different messages to process.

- Reference:
  - https://redis.io/topics/streams-intro
  - https://blog.logrocket.com/why-are-we-getting-streams-in-redis-8c36498aaac5/

Info

Scan

Sorting

Common cache problems

Cache big values

hot spot