Solution

Step 1: Gather the requirements

Functional Requirements

a. cacheValue get(key)
b. put(key, value)

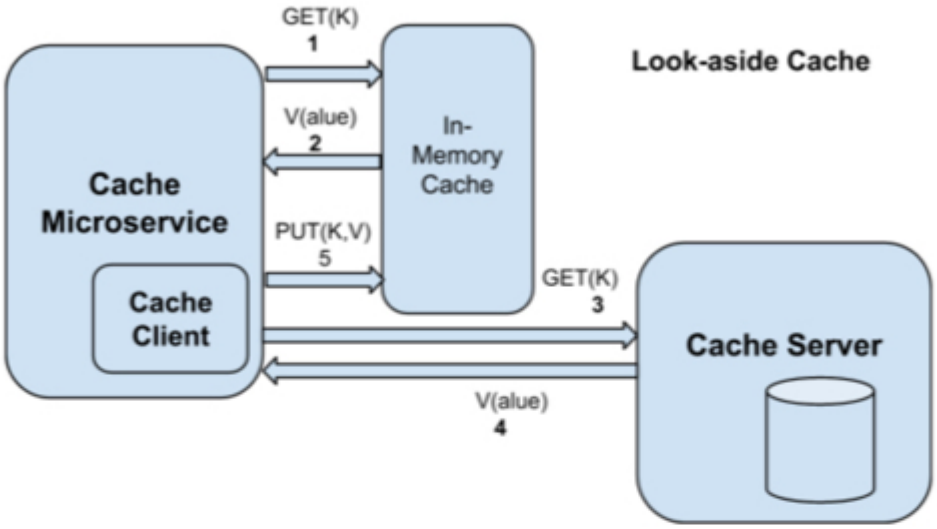
Non-Functional Requirements

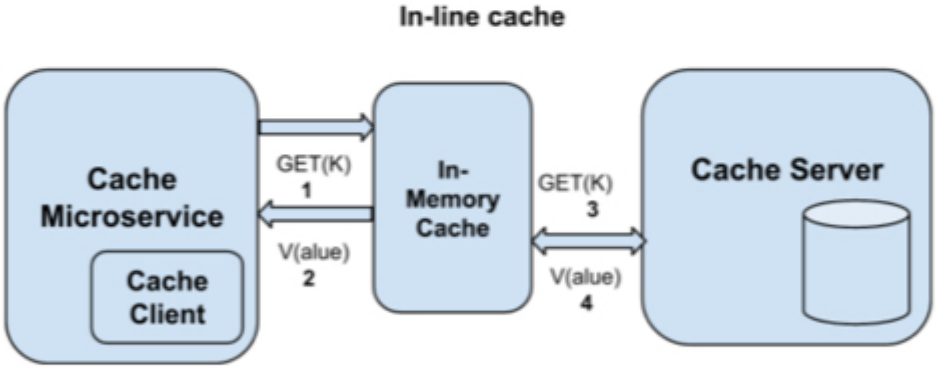
a. The cache solution must scale as user base increases
b. The solution must be highly available for most of the times (99.9 (3 9s'))
c. Minimal Latency (upto 100ms)

Step 2: Define Microservices

Microservice	**Type of Technology**
CacheService microservice	K-V Workload with in-memory Hashmap

Step 3: Draw the Logical Architecture





Outline

Type of Cache	Read	Write	Type of Application Applicable to
Look-aside Cache	1. Application requests value for a key 2. In-Memory Cache returns with the	The Cache microservice writes new data to the cache server database and	1. This strategy is used for in-memory cache data that does not change often. If the

	value requested for the key 3. If value (for the associated key) is not present, then the Cache Microservice fetches the value from the Cache Server Database (**read-aside)** 4. The server responds with the associated value for key 5. The Cache microservice then does a Put (Key,Value) into the in-memory cache by deleting the previous entry.	then updates the in-memory cache, after invalidating the cache.	in-memory cache data changes often, then the cache miss / invalidation would diminish the returns of the in-memory cache strategy.
Inline Cache	1. Application requests value for a key 2. In-Memory Cache returns with the value requested for the key 3. In-memory cache retrieves data from the cache Server database (**read-through)** 4. The response to the request is updated in the in-memory cache and responded to the application	Application writes new data to the in-memory cache The in-memory cache will write synchronously (**write-through**) or asynchronously (**write-behind**) to the cache server.	

Two basic operations are supported by the Cache microservice:

a. get(Key), given a key, the associated value(s) to be fetched. As outlined above, multiple strategies are possible to achieve this.
    i. Maintaining an in-memory cache within the cache microservice. If the request can be satisfied by the in-memory cache content, the value associated with the key is returned as response.

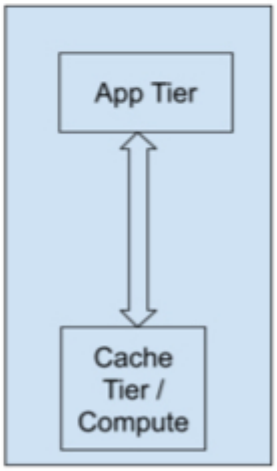
ii. If the key is not present in the in-memory cache, this results in a cache miss. This will result in querying the **Cache Server**. If the strategy follows, **look-aside** cache strategy, the **Cache Server** is queried for the value of **cache-missed** key.

The **Cache Server** responds with the associated value stored in the database.

b. put(Key, Value), This method will either add a <Key, Value> pair to the cache or update a value to an existing Key. When adding a Key, two cases are possible:
   i. The addition of <Key,Value> is successful, as there was enough space in the cache to accommodate this request.
   ii. There is not enough space in the cache for adding this entry. The entries in the cache need to be evicted. **Least Recently Used (LRU)**, is one of the popular approaches for evicting a cache entry, wherein an entry that has been the oldest entry will be evicted or removed from the cache, to make way for adding a new entry.
   iii. It should also be noted that even though the cache size is sufficient for adding new entries, oldest entries should still be evicted from the cache. This entry may not be accessed and still be present in the cache. Such entries need to be removed too. The strategy that needs to be followed is add a **Time To Live (TTL)**, to each of the cache entries and automatically evict them on timer expiry. We need to come up with a value which is neither too aggressive nor too lenient.

Step 4: Deep Dive into Microservice

Cache Microservice



On a single server multi-threaded environment wherein it is possible that multiple threads could update a single key. In such situations how should the system ensure that the values associated with the key remain consistent.

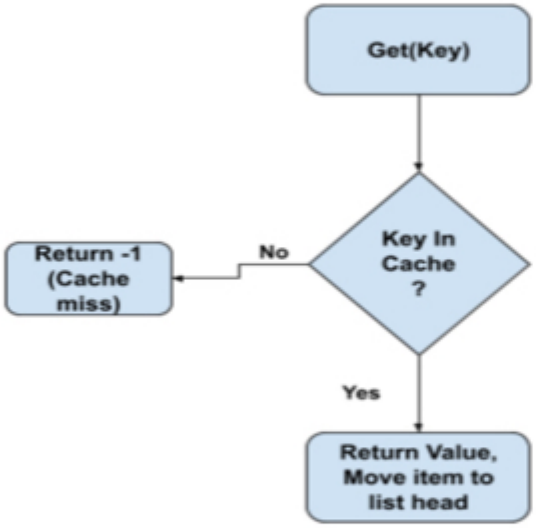
To avoid the situation known as “Last Write Wins” where multiple instances update the key concurrently, safeguards can be put in place;

a. The system can implement a CAS, (Check And Set) way for updating the keys when multiple threads modify the same key.
b. Typically, a **Watch** can be initiated for the key that needs to be monitored for changes across multiple threads.
c. The key can be updated
d. The update can then be propagated to the In-memory Cache.
e. In case before the update being performed, any other thread(s) modify the same key, then the whole transaction can be aborted.

Cach Initialization

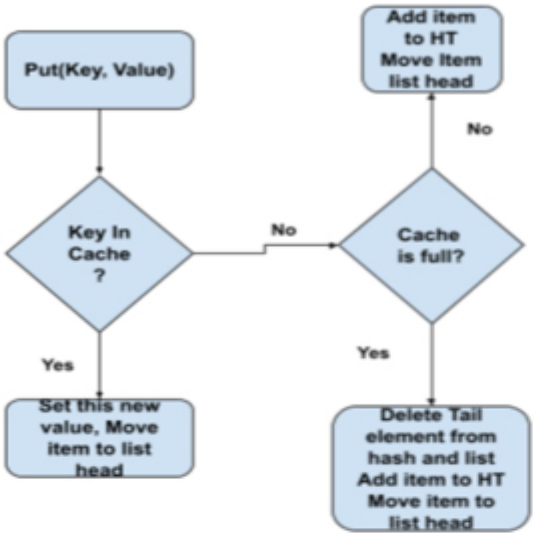
```
class LruCacheNode:
      def __init__(self, next=None, prev=None, self.key=key,
self.value=value)

class LruCache:

      def __init__(self, capacity: int):
          self.capacity = capacity
          self.cacheMap = dict()
```

Get Operation




```
def get(self, key: int):
    if key not in self.cacheMap:
        return -1 // Cache miss
    else:
        extractAndRemove(key)
```

Put Operation




```
def put(self, key: int, value )-> None:
    // Key exists in cache
    if key n self.cacheMap:
         self.cacheMap[key].value = value
         extractAndRemove(key)

     // Key does not exist
     if len(self.cache) == self.capacity:
        del self.cacheMap[tail.key]
        penultimate = self.tail.prev
        if penultimate is not null:
            penultimate.next = null

        self.tail = penultimate
        len(self.cache) -= 1

     if head is null:
         head = tail = Node(key, value)
     else:
         newNode =  Node(key, value)
         newNode.next = head
         head.prev = newNode
         head = newNode
```

```
    self.cacheMap[key] = head
    len(self.cache) += 1

def extractAndRemove(key: int)->None:
    node = self.cacheMap[key]

    if head == node:
       return
    prevnode = node.prev
    nextnode = node. Next

    // Adjust pointers now
    prevnode.next = nextnode

    if nextnode is not null:
        nextnode.prev = prevnode
    else:
        tail = prevnode

    node.next = head
    head.prev = node
    head = node

    node.prev = null
```

Time Complexity

All the insert key, update key, search key operation runs in O(1) time, hence the time complexity for both put and get methods runs in O(1) time

Space Complexity

As for the space complexity, the hash table uses O(n) and each entry in the hash table has n linked list entries. So the overall space complexity is O(n+n) = O(2n) = O(n)

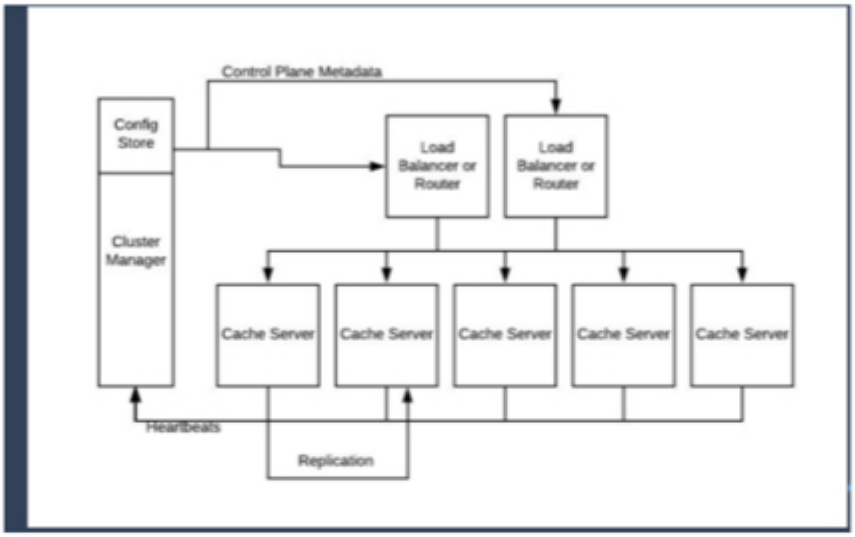
Step 5: Identify need for Scale

Service	Scalability Need	Scalability Dimension
Cache Microservice	• Shard Replication • Hotspots avoidance • Geographic Distribution • Throughput • Latency	Horizontal Scaling

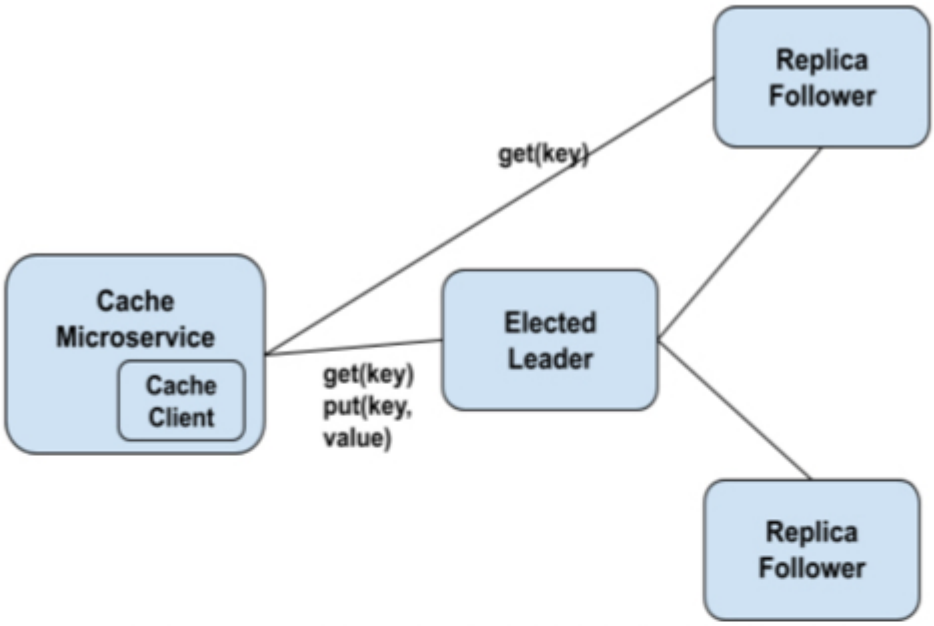
Scalability Feature	How achieved	Component
Throughput	Replication 3X Sharding	a. Database (Horizontal Sharding based on Key, Consistent Hashing for servers)
Hotspot Avoidance	Load-Balanced Servers Sharding	a. Replica Follower
Availability	Replication	a. Replica Follower
Consistency	Eventual Consistency	a. Elected Leader node and Replica Follower
Latency	Caching	

Step 6: Propose Distributed Architecture

Distributed Architecture for Cache Layer



a. Cache Client part of the Cache Microservice contacts the Config Store for the list of cache servers. These can be searched using Binary search in O(log n) time
b. Consistent hashing is the preferred way for hashing the Cache servers. Smaller fraction of keys will be re-hashed when a new cache server is added or removed from the cluster.
c. The Configuration Store holds the information about availability of cache servers and makes them unavailable based upon heart-beats that are received from each of the cache servers.
d. When a cache miss happens, it is concluded that the cache server is offline or unavailable.



Replication shall provide high availability and hotspots. Replication can be done by Leader-Follower configuration as shown above. Replica followers can be horizontally scaled if hotspots are still found. All the **put** operations will go to the Leader node and the **get** operation will be handled by both the leader as well as replica slaves. The **Configuration Service** can provide leader election among a set of nodes and track the health of the nodes. On failure of the leader node, the Configuration service can elect a new leader and convey that information to the Cache clients.