System Design

Graphical user interface, application

Description automatically generated

[A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.]

**User Datagram Protocol (UDP)**

UDP is connection less. Datagrams are guaranteed only datagram level. Datagram might reach their destination out of order or not at all. UDP does not support congestion control. Without guarantees that TCP support. UDP is generally more efficient.

UDP can broadcast, sending datagrams to all devices on the subnet. This is useful with DHCP because client has not received IP address, thus preventing a way TCP to stream without the IP address. UDP is less reliable but works well in real-time use case such as VoIP chat streaming and real-time video call.

Dynamic Host Configuration Protocol (DHCP) is **a network protocol that is used to configure network devices to communicate on an IP network**. A DHCP client uses the DHCP protocol to acquire configuration information, such as an IP address, a default route, and one or more DNS server addresses from a DHCP server.

**UDP:**

* User Datagram Protocol plays an important in computer networks in speeding up the efficiency of the network on websites that consume a lot of data.
* UDP is less secure than TCP but on the other hand, it is very much faster.
* UDP technically is independent of the source and destination of the end-to-end transfer of data packets using the “three-way handshake technique.”
* The main purpose of UDP is for fast and reliable internet application usage and data transfer and doesn’t require data verification and security services.
* UDP has less bandwidth and latency which makes it work even in terrific network connections.
* When a large number of users the information at a particular time, so we need to manage the traffic and show responses to their queries, we use UDP in areas like gaming, video streaming, and online meetings.

**TCP:**

* The basic use of TCP is to deliver data packets from the server to the target computer.
* TCP works on the overhead of IP to ensure bulk internet connectivity and transfer them in the correct order and also, they get delivered in the same order.
* TCP lags in latency and data fluctuation on the data-consuming applications.
* The most important feature of TCP is that it maintains its large focus on data security and verification options.
* When UDP sends data packets process-to-process messages which are not secure, TCP uses host-to-host messaging to enable a safe and secure environmental network.
* TCP is reliable for online streaming services as the customer’s data is completely secure and kept in safe hands.
* TCP enables IP header to produce error-free and corrupt-less data in an ordered manner, so that receiver does not face any issue.

Table

Description automatically generated

Table

Description automatically generated

1. [LRU Cache](https://leetcode.com/problems/lru-cache/)

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

We asked to implement which provides, following operations in O(1) time.

* Get the key/Check if the key exists
* Put the key
* Delete the first added key

The first two operations O(1) time are provided by the standard HashMap and the last one – by link list

There is a structure called ordered dictionary, it combines behind both HashMap and link list.

Called LinkedHashMap.

**Time Complexity:** O(1) both for put and get since all operations with ordered dictionary:

Get/in/set/move\_to\_end/popitem

(get/containsKey/put/remove) are done in constant time.

Space Complexity: O(capacity) since the space is used only for an ordered dictionary with at most capacity + 1 elements.

**Approach 2: HashMap + DoubleLinkList**

**Graphical user interface, text

Description automatically generated**

One interesting property about the double link list is that node can remove itself without reference. In addition, it takes constant time to add and remove nodes from the head or tail.

One particularity about the double linked list implemented here is that there are pseudo tail to mark the boundary, so that we don’t need to check the null node during the update.

Create a inner class with name DLinkedNode and add two references variable to the same class, as prev and next to represent double linked list. Also add key and value variable.

In LRUCache class, add hashmap key as Integer and value as DLinked  
Node.

Create a variable called size to track number of element in cache, and capacity and two Dlinkled class variable names as head and tail

Inside LRUCache Construcutor , initialized capacity , size and prev and next to new DlinkedNode.

Once object created

Diagram

Description automatically generatedText, whiteboard

Description automatically generated

Diagram

Description automatically generated

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

Text, whiteboard

Description automatically generated

Text

Description automatically generated

1. [Implement Stack Using Queues](https://leetcode.com/problems/implement-stack-using-queues/)

Graphical user interface, text, application

Description automatically generated

1. [Implement Queue using Stacks](https://leetcode.com/problems/implement-queue-using-stacks/)

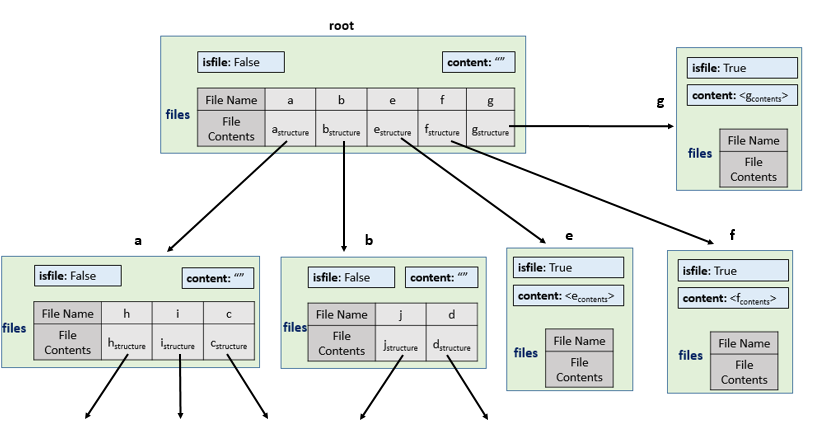
Graphical user interface, text, application

Description automatically generated

1. [Design In-Memory File System](https://leetcode.com/problems/design-in-memory-file-system/)

Approach: Using unified directory and File List

Data structure for a directory contains a unified files HashMap, which contains the list of all the files and subdirectories in the current directory. Apart from this, we contain an isfile, which when True indicates that the current files entry is actually corresponding to a file, otherwise its represents a directory. Further, since we are considering the directory and files entries in the same manner, we need an entry for content , which contains the content file (if isfile entry is True in the current case). For entries corresponding to directories, the content field is kept empty.



Text, letter

Description automatically generated

A picture containing text

Description automatically generated

1. [Insert Delete GetRandom O(1)](https://leetcode.com/problems/insert-delete-getrandom-o1/)
2. [Design a Text Editor](https://leetcode.com/problems/design-a-text-editor/)

**Always maintain the left part of string in left stack and right part of the string in right stack which are divided by the cursor**

* **Add operation:** Push string character by character to the left stack.
* **Deletion operation:** Pop characters from the left stack and return the number of characters popped.
* **Cursor movement to left:** Pop elements from left stack and push in right stack.
* **Cursor movement to right:** Pop elements from right stack and push in left stack.

**TBD**

1. [Rate Limiter](https://leetcode.com/discuss/interview-question/system-design/1616482/System-Design%3A-Rate-Limiter#:~:text=Rate%20Limiter%3A%20A%20rate%20limiter,Password)%20once%20every%205%20minutes.)

Design a Rate Limiter: A rate limiter in HTTP world is used to limit the number of client requests allowed to be processed over a specific period.

* In a banking system you may request OTP (One Time Password) once every 5 minutes.
* In a social media platform, a user maybe allowed to post maximum 5 posts per minute
* You can register a new user from same IP maximum 5 times per day. This is case where we want to prevent bot/sock-puppet accounts

Benefits of having a Rate Limiter:

* Prevents DDoS/DoS (Denial of Service) attacks. DDoS attacks can swamp the system and cause resource starvation leading to massive downtimes. A proper and robust rate limiter can help reduce chances of DDoS attack being successful
* Cost benefit: If you have a rate limiting system in place, you can control maximum requests that your system is allowed to process. This is turn can be used to allocate resources in other places rather than scaling servers to handle ever increasing request per second

In general, a system design interview goes as below

* Gather scope requirements - High level idea on what is the intent, scope, scale of system. Standalone vs BeSpoke service etc. Distributed vs non distributed. This is important because it helps us design keeping the scalability aspect in mind
* Functional and Non functional requirements: Functional requirements pertains to the APIs of your system and Non Functional refers to the performance aspect such as availability, low latency etc
* Basic design - start with a simple High level diagram (a few boxes is fine) explaining the flow and various helper services you can think of. Here it is important to ask interview as to where they want you to deep dive. For example, if you are desigining a ride-share service (Uber) then it depends on what interviewer wants to focus on, maybe they are interested in driver pricing optimisation or they want to improve the map search portion. Clarify whenever you are uncertain. System deisgn interviews are DISCUSSIONS with NO PERFECT SOLUTION.
* Dive deeper as per instructions and expand your design, it is a good practice to highlight possible shortcomings/bottlenecks in your design while you are making them.
* Validation: validate that your design works, address possible bottlenecks and seek interviewer feedback
* Future scope: such as a metric service to measure performance of your design; whether your design will work with 1 million customers, 100 million customers etc. What may work for 1 million users may not scale for 100 million.
* closing points: aligns with point 6 but can be a general commentary on your design and you may also mention other solutions briefly

Now lets us proceed to the actual solution portion:

Now let us proceed to the actual solution portion:  
We can ask a few questions such as, **If the rate limiter is to be a server side/client side?**, **Whether we should inform users/clients who are rate limited?**, **What is the expected scale of the system. How much throughput are we looking forward to handle?** and **Whether rate limiting is client specific or system specific, like if we would use IP, userId etc to enforce the rate limiting logic**

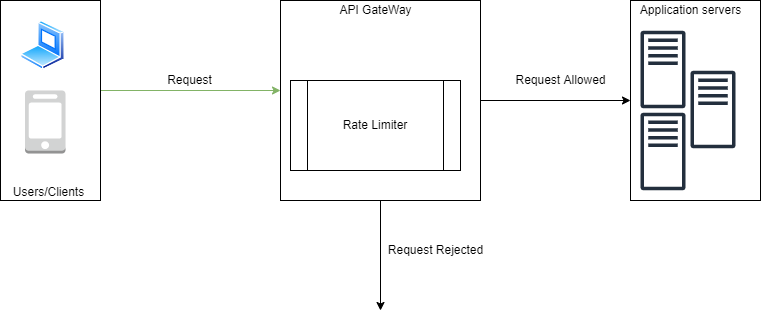
Based on the above I am assuming below as my **functional requirements**:

1. Server-side rate limiting
2. Should be able to handle large number of requests
3. Clients should be informed of being rate limited (proper exception handling)
4. For now, let us assume rate limiter is a separate service
5. should have clearly defined APIs such as allowRequest() that would return a boolean signifying if request is to be processed or dropped

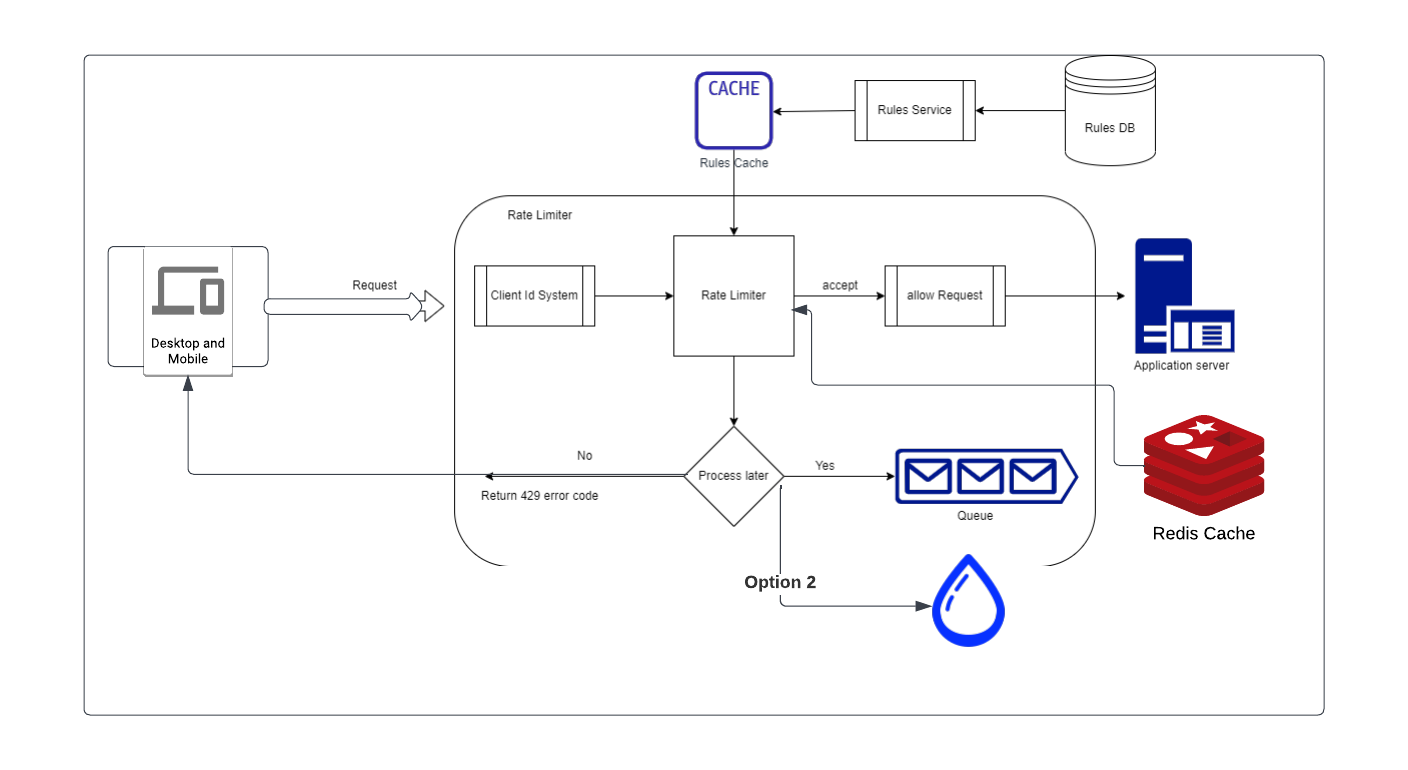
**Non-functional requirements**

1. Highly available service (since our system is a standalone application that will be used by various other services, it should be highly available)
2. Low latency: The rate limiting should not impact the application's response time. Our logic should be efficient and optimized
3. Fault tolerant: if rate limiter is down, it should not cause failures on the client applications. Failures needs to be handled
4. Resource optimisation: Rate limiter should not be too expensive to implement

A very simple design to start with:



Now we can proceed further, at this point we can explode the rate limiter component and go to full-fledged design. In order to do that we must keep track of the **functional** requirements



Briefly let us go over the job/role of each component

* Client ID system is for helping us identify client. As our rate limiter is plug and play component, it must have a mechanism to id the clients and then use the id to pull the rules.
* Rules DB is to store rules for all clients onboarded onto our service.
* Rules Service is responsible for loading Rules from DB into Rules Cache
* Rules Cache is for fast access of rules for client using their id (coming from id system)
* Rate limiter, it fetches counters and last request timestamp from Redis cache. Based on the response, the rate limiter decides
  + If the request is not rate limited, it is forwarded to API servers.
  + If the request is rate limited, the rate limiter returns 429 too many requests error to the client. In the meantime the request either dropped or forward to the queue.

Following important rate limiting algos

* Token Bucket (quite common and very easy to implement)
* Leaking bucket
* Fixed Window
* Sliding Window
* Sliding Window logged

Validation (that our design works)  
Scenario 1: Request comes => client Id is say 100 => Rate Limiter pulls rules for id 100 from Rules Cache => Rate Limiter validates if enough tokens are there for request to pass => tokens are there => request allowed and sent to Application server

Scenario 2: Request comes => client Id is say 100 => Rate Limiter pulls rules for id 100 from Rules Cache => Rate Limiter validates if enough tokens are there for request to pass => not enough tokens => request rejected and HTTP code 429 (too many requests) returned to client.

Questions for you.

1. How will you make it distributed?
2. what challenges you forsee if distributed Rate limiter is implemented.
3. Can you identify points of failures in this design.
4. Handling failure when Rate Limiter service goes down
5. Rate Limting done at other level of the OSI model