Problem Statement

Your team has been working on an online e-commerce website at ABC Company. Suddenly, the website faces a large number of requests in a short period of time, exceeding the system capacity, which leads to Flooding. These flood attacks cause the system to receive too much traffic for the server to buffer, causing them to slow down and eventually stop. The application’s target users are also experiencing errors about “too many connections” or something similar when visiting a website or using the app. You, along with the team, urgently need to solve this significant security issue. What should be the solution to this problem? Think about limiting the user requests and setting a limit. What will the advantages of such a system be?

# Objective of API rate limiter

1)Providing security against DOS attacks

Let us say there is a malicious user who is calling your API serveral times such that it is blocking the incoming requests for any other users such attack is known as Denial of Service (DOS) attack.

2)Resource management

Once we have estimation of the load from the users, we can predict the required and enough resources.

3)Business Use case Analysis

It provide analysis of uasage of each user.

# Features

## Functional-requirements

* We should be able to limit the number of requests that user can send to an API to a certain limit.
* Let us say 15 requests/minute
* If the user exceeds the rate limit, he should be getting the 429 (Too Many Requests response status code)

## Non Functional-requirements

No significant latency added as part of our limiter service.

Our service should be available 99.9% and responding with valid response all time.

# Estimations

Let us say that traffic is 1000 requests/seconds (all kind of users)

Let us assume 1 M users accesing our service per day.

# High Level Design

User/Client request will first reach the webserver (exclusively handles HTTP requests).

Webserver will contact our API rate limiting service.

Rate limiting service will use its own DB or Cache and its businesslogic to determine if the request is valid or invalid.

If the request is valid, webserver will hit the application server (serves business logic to application programs), otherwise it will go back to the client with response code of 429.

Rate limiting service can be maintaained as a microservice, there are few advantages there

1)Independent code management

2)Independent scaling

3)SPOF: let us say the API limiter service goes down, we need to define the rules which can pass all the requests to the application server. even though there is over utilizaton of APIs, server will still respond.

## Rate limiting algorithms

### Fixed window algorithm

Maintain the time windows to accept or reject an incoming request.

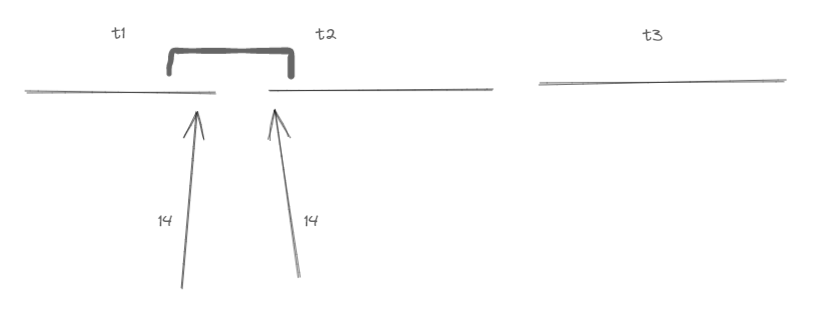
Algorithm to be used here is fixed time width windows , we will use system clock to take 1 min intervals as fixed time width for any user.

All users have same common time window of 1 min interval.

There is a problem with this approach,let us take the time interval limit to be 60 seconds and user A who has made the flood of request at end of t1 , and beginning of t2 14 req each.

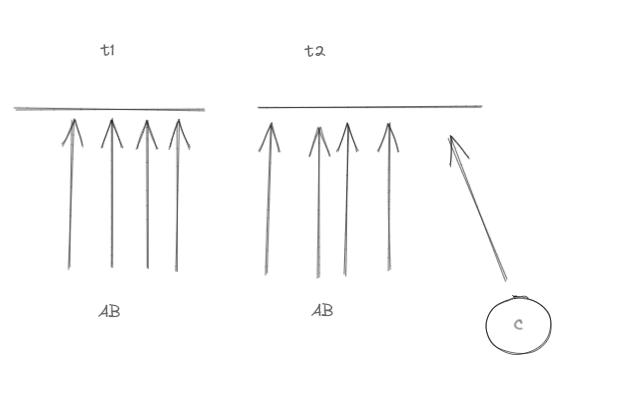
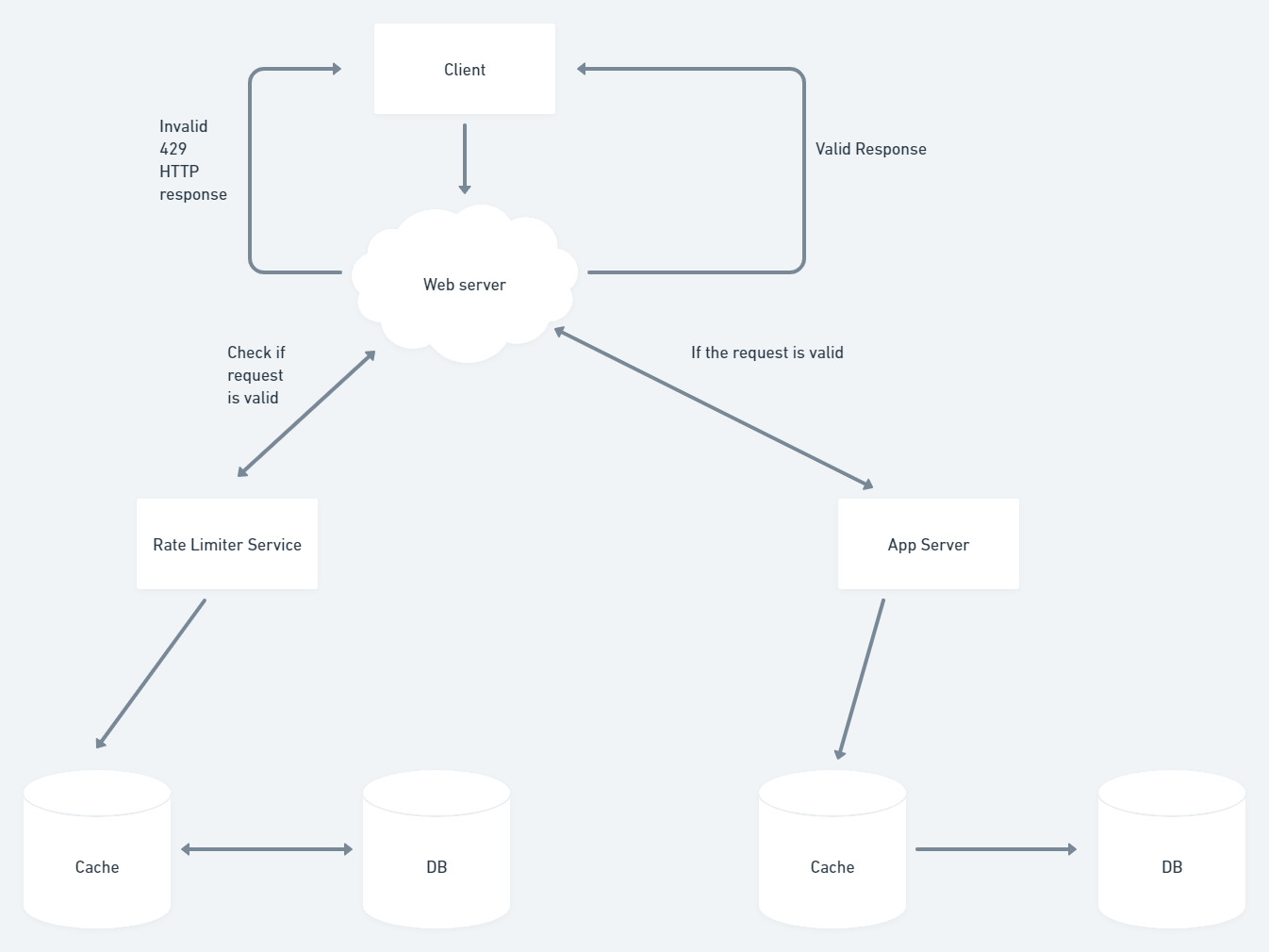
That is still within the threshold , but if we see within that 2 or 3 seconds there are more requests than the threshold still all of them are valid.

This is issue with bursts of requests at the end or the start of the interval.



The second problem is let us say that users A and B are making the continous requests , this might result in the choke of service for any other user C.

This will lead to the starvation of resources for C.



### Sliding window Algorithm

For each user request maintain a hash table key is userId and values are sorted set of timestamps of requests.

userId : {} -->timestamps of requests in sorted order

this hash table is maintained in the cache of the hashtable.

Whenever a request comes from a userId with timestamp t

1)We’ll remove all the timestamps that are older than current\_time(t)-1 min(the window size)

i.e t-1 .

2)Check the length of the sorted set for that particular userId.

If len(sortedSet)> k (threshold i.e., here 15 requests)

We need to reject the current request.

Otherwise, can accept the current request.

3)After that we have computed this we need to insert the current timestamp to the set .(for accepted req).

4)The accepted req will get forewarded to the App server. rejected req can get 429 responses.

Memory req

UserId: {}

Each user Id --> 10 bytes

Each timeStamp --> 10 bytes

Max len of sorted set --> 1000

Max size of sorted set --> 1000 x 10

For 1 user --> 10 + 1000 x 10 ~ 10 kb

For 1 M users --> 1 M x 10 kb ~ 10 GB

# Scaling

## Sharding

We’ll be storing the userId: {} hashtable.

UserId can be i) Auto-increment or ii) Alpha-numeric which is a good sample for Sharding.

We can shard the cache based on userId.

Or we can also shard the appservers based on the most frequent APIs.

## Replication

We can replicate the cache server for the Rate limiter server.

If the default cache server goes down, we can use alternative replica server (fault tolerance)

They have a heart-beat mechanism (which will ensure that replicas are in sync)

## Caching

(In a different sense in terms of the frequency of users)

Let us say that 20% of users are accessing the APIs more frequently than others, they are the high value users.

We can have separate cache servers for this 20%, so that we can have low latency for these particular users.

These servers can be placed at the geographical locations close to those users.

So that that other users will not get starved because of the high frequency of these users.

Cache eviction policy:

We can use the LRU policy for eviction when the cache is full.

## Horizontal Scaling

Load on the rate limiting server will be more compared to that of the app server becz only the accepted request will go through the app server, no matter what all req should go through the rate limiting server.

So, we have to horizontally scale the rate limiting server.

Instead of having 16 GB single rate limiting server,we can have 4 x4 GB horizontal servers.

## Load balancing

Since we have scaled the rate limiting server , we need to place a LB inbetween the web server and rate limiting servers.

We can use the RR algorithm for equal distribution of the load.